

# Introduction to Deep Learning

Giovanni Della Lunga

[giovanni.dellalunga@gmail.com](mailto:giovanni.dellalunga@gmail.com)

ADVANCED TOPICS IN ARTIFICIAL INTELLIGENCE

Bologna March 04-22, 2024

# We will talk about...

In this lesson, we will cover the following topics:

- Introduction to Deep Learning. We will start by introducing the concept of deep learning, its history, and its applications.
- What is a Neural Network? We will delve into the fundamental building block of deep learning, the neural network. We will cover the architecture of a neural network and its different components.
- The Feed Forward Architecture. We will explore the simplest type of neural network, the feed-forward architecture, and learn how it can be used to solve a variety of problems.
- We will introduce Keras, a popular deep learning library, and learn how to use it to build and train neural networks.
- We will cover the basics of sequential data, such as time series, and explore how neural networks can be used to model and predict such data.

# What is Deep Learning?

# Introduction: Machine Learning and Deep Learning

- A deep-learning model transforms its input data into meaningful outputs, a process that is **learned** from exposure to known examples of inputs and outputs.
- Therefore, the central problem in deep learning is to **meaningfully transform data**: in other words, **to learn useful representations of the input data at hand, representations that get us closer to the expected output**.
- What is a **representation**? At its core, it is simply a different way to look at data, to represent or encode data.

# Introduction: Machine Learning and Deep Learning

The central problem in deep learning is to **meaningfully transform data**

# Machine Learning and Deep Learning

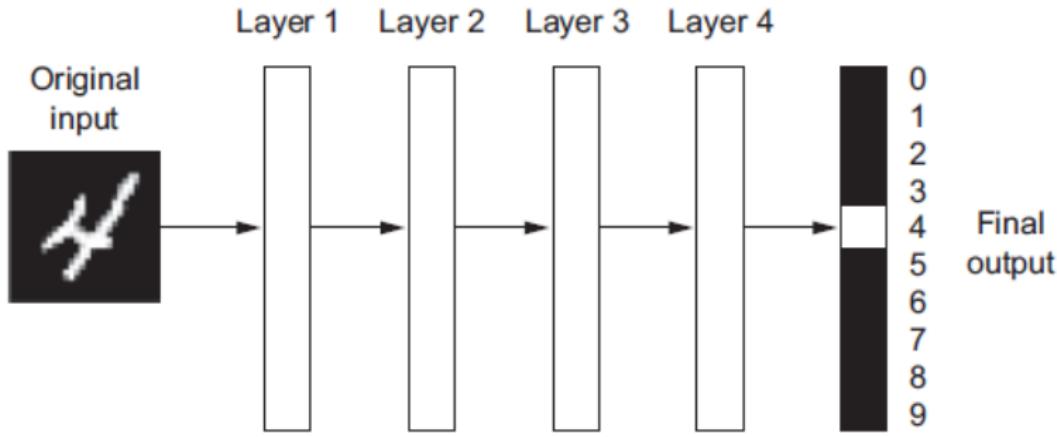
- From a more formal point of view, deep learning learning process involves input variables, which we call  $X$ , and output variables, which we call  $Y$ .
- We use neural network **to learn the mapping function** from the input to the output.
- In simple mathematics, the output  $Y$  is a dependent variable of input  $X$  as illustrated by:

$$Y = f(X)$$

Here, our end goal is to try to **approximate the mapping function**  $f$ , so that we can **predict** the output variables  $Y$  when we have new input data  $X$ .

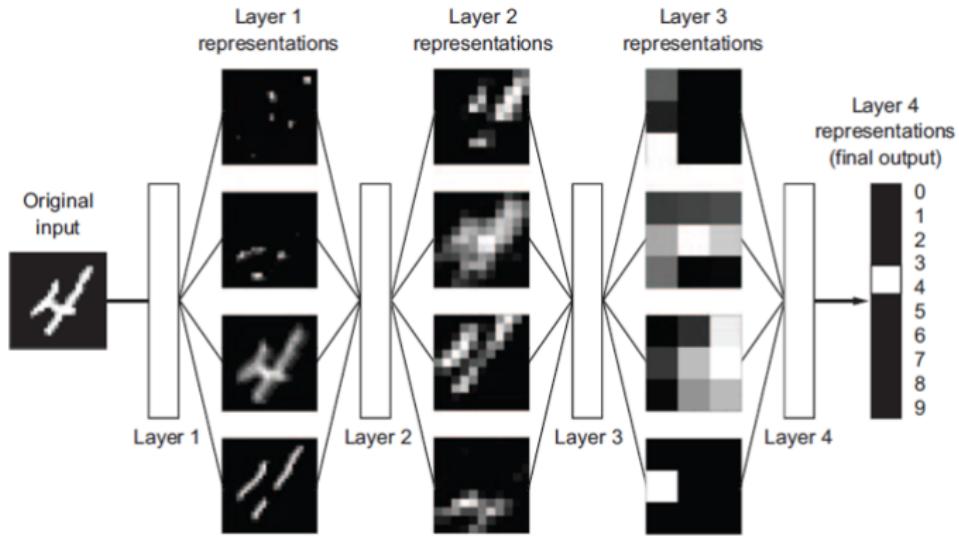
# Introduction: Machine Learning and Deep Learning

Neural networks are typically composed of several layers of interconnected nodes, with each layer performing a specific transformation on the data. The input data is fed into the first layer, and the output of that layer becomes the input to the next layer, and so on, until the final output is produced...



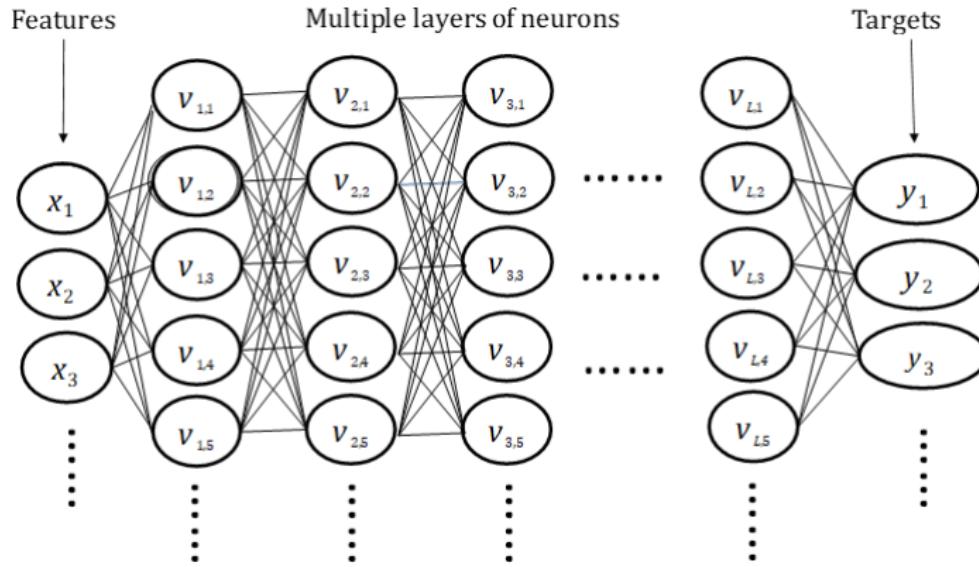
# Introduction: Machine Learning and Deep Learning

Neural networks are typically composed of several layers of interconnected nodes, with each layer performing a specific transformation on the data. The input data is fed into the first layer, and the output of that layer becomes the input to the next layer, and so on, until the final output is produced...



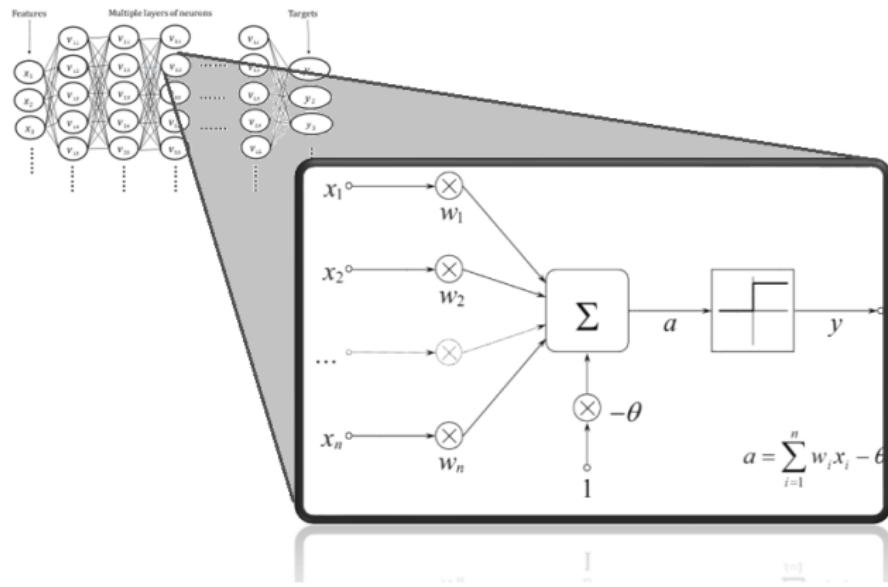
# Introduction: Machine Learning and Deep Learning

Neural networks are typically composed of several layers of interconnected nodes, with each layer performing a specific transformation on the data. The input data is fed into the first layer, and the output of that layer becomes the input to the next layer, and so on, until the final output is produced...



# Introduction: Machine Learning and Deep Learning

Neural networks are typically composed of several layers of interconnected nodes, with each layer performing a specific transformation on the data. The input data is fed into the first layer, and the output of that layer becomes the input to the next layer, and so on, until the final output is produced...



# Neural Networks

## Subsection 1

### The McCulloch-Pitts Neuron

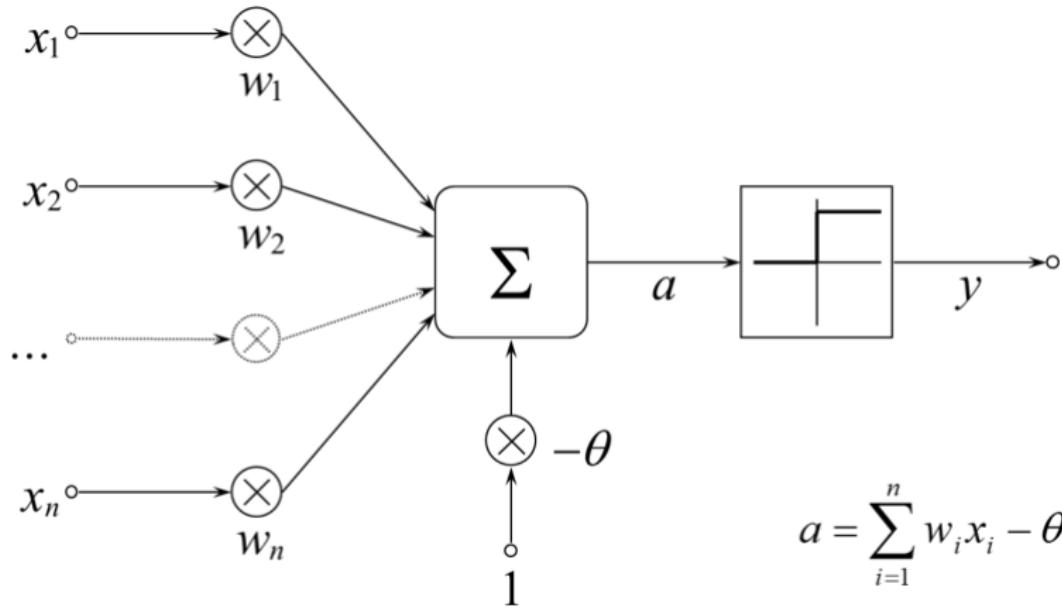
# Mc-Culloch and Pitts Neuron

- The McCulloch-Pitts neuron, also known as the binary threshold neuron, is a simple mathematical model of a biological neuron that was introduced by Warren McCulloch and Walter Pitts in 1943. It is one of the earliest and most fundamental models of artificial neural networks.
- The McCulloch-Pitts neuron is a computational unit that takes in one or more binary inputs, sums them up, and applies a binary threshold function to produce a binary output. The threshold function is such that if the weighted sum of the inputs is greater than a certain threshold value, the neuron outputs a 1 (i.e., it fires), and otherwise, it outputs a 0 (i.e., it remains inactive).

# McCulloch and Pitts Neuron

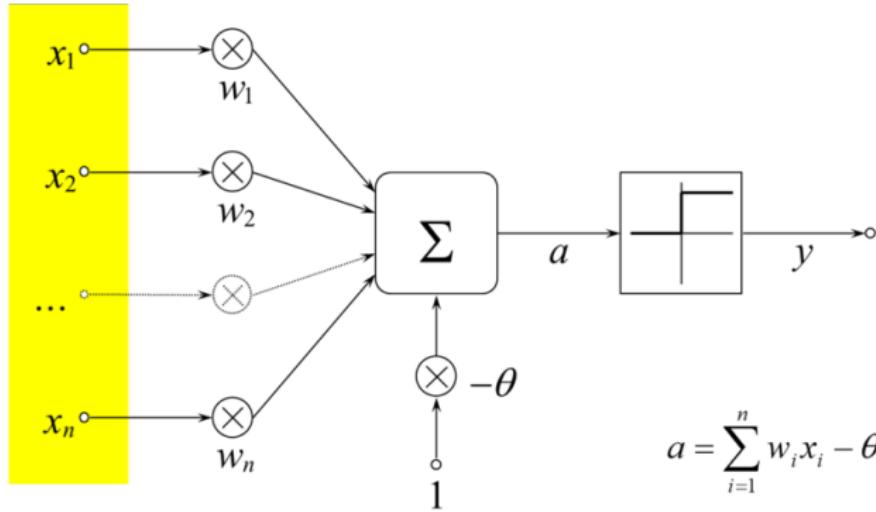
- The McCulloch-Pitts neuron can be used to model basic logic operations, such as AND, OR, and NOT, and can be combined in various ways to build more complex computational systems. While the McCulloch-Pitts neuron is a highly simplified model of a biological neuron, it serves as a foundation for more sophisticated neural network models, such as the perceptron, which was introduced a few years later by Frank Rosenblatt.

# McCulloch and Pitts Neuron



# NN Data Flow

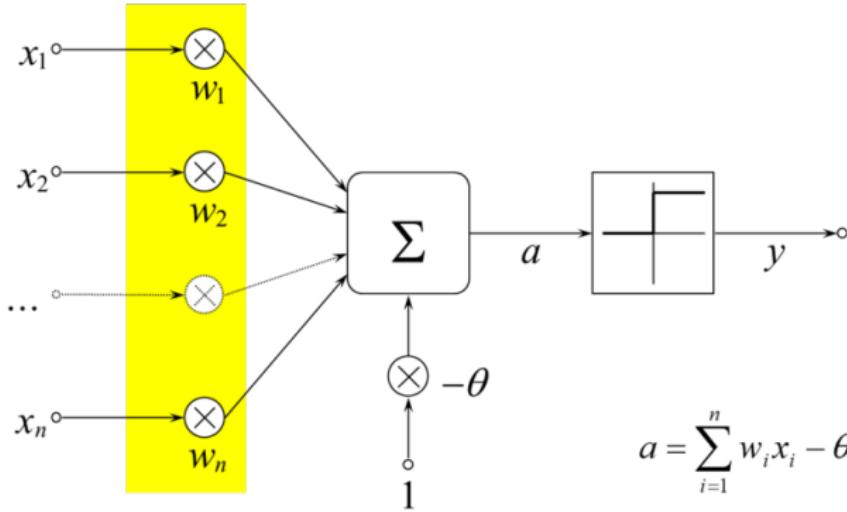
## McCulloch and Pitts Neuron



INPUT DATA	WEIGHTS	WEIGHTED INPUT	BIAS	ACTIVATION FUNCTION	OUTPUT
------------	---------	----------------	------	---------------------	--------

# NN Data Flow

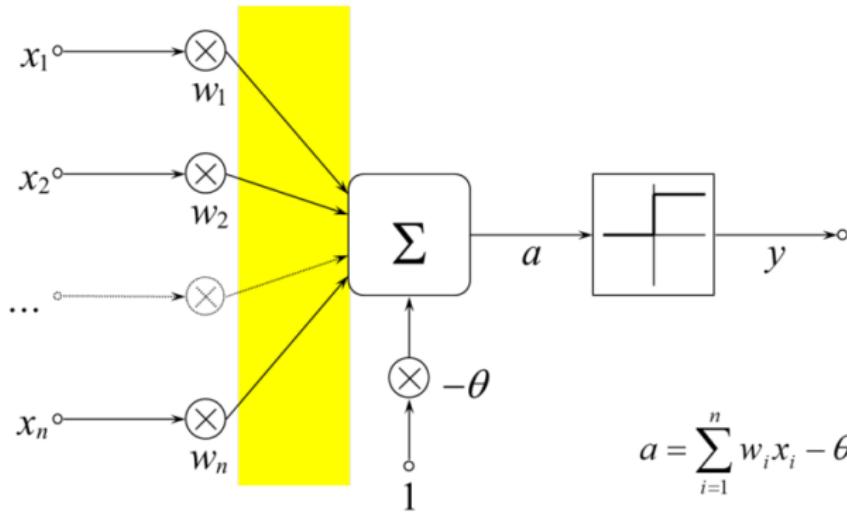
## McCulloch and Pitts Neuron



INPUT DATA **WEIGHTS** WEIGHTED INPUT BIAS ACTIVATION FUNCTION OUTPUT

# NN Data Flow

## McCulloch and Pitts Neuron



INPUT DATA	WEIGHTS	WEIGHTED INPUT	BIAS	ACTIVATION FUNCTION	OUTPUT
------------	---------	----------------	------	---------------------	--------

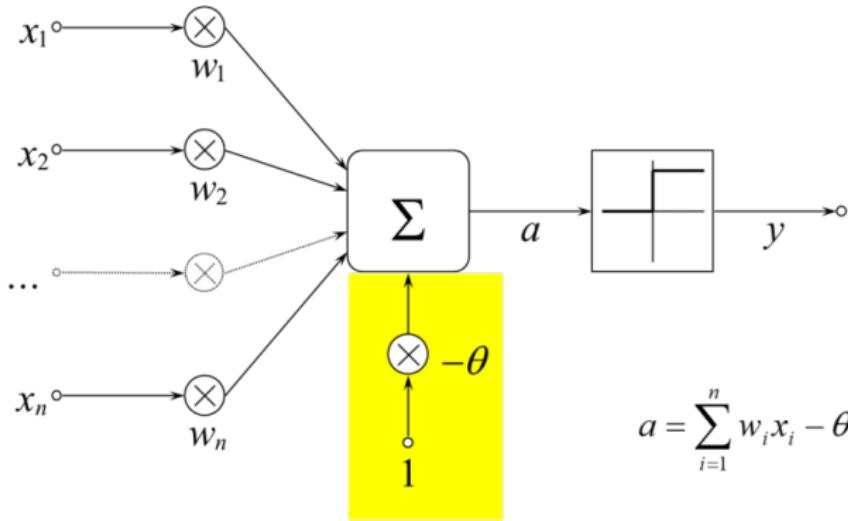
# Mc-Culloch and Pitts Neuron

From a functional point of view

- an input signal formally present but associated with a zero weight is equivalent to an absence of signal;
- the threshold can be considered as an additional synapse, connected in input with a fixed weight equal to 1;

# NN Data Flow

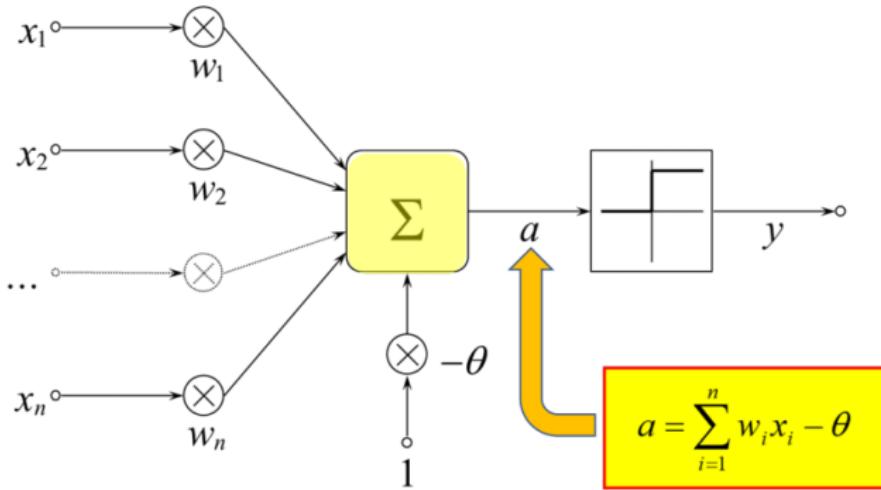
## McCulloch and Pitts Neuron



INPUT DATA WEIGHTS WEIGHTED INPUT BIAS ACTIVATION FUNCTION OUTPUT

# NN Data Flow

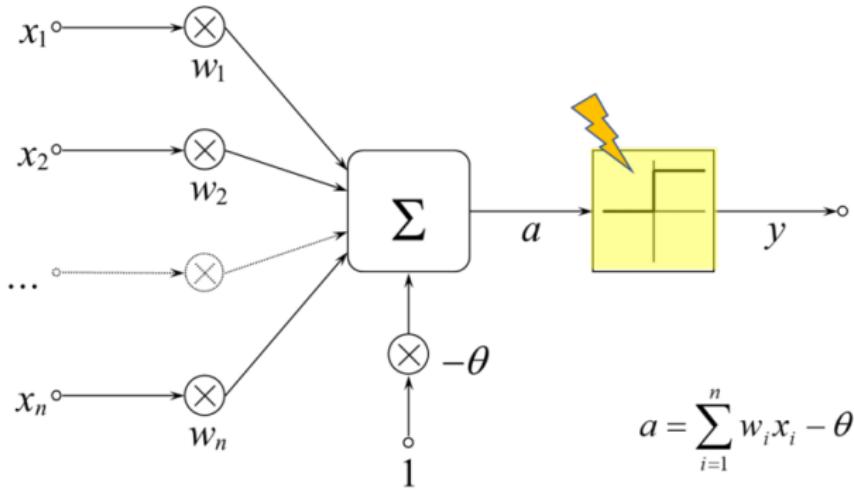
## McCulloch and Pitts Neuron



INPUT DATA WEIGHTS WEIGHTED INPUT BIAS ACTIVATION FUNCTION OUTPUT

# NN Data Flow

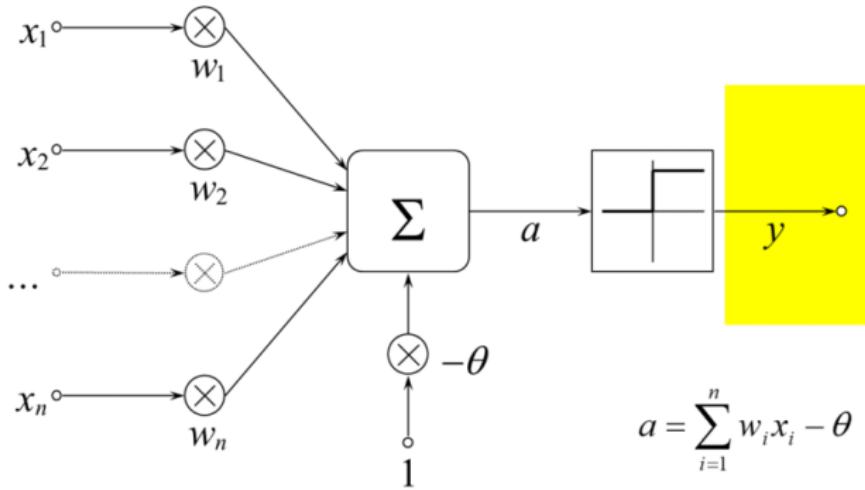
## McCulloch and Pitts Neuron



INPUT DATA WEIGHTS WEIGHTED INPUT BIAS ACTIVATION FUNCTION OUTPUT

# NN Data Flow

## McCulloch and Pitts Neuron



INPUT DATA WEIGHTS WEIGHTED INPUT BIAS ACTIVATION FUNCTION OUTPUT

# Activation Function

$$a = \sum_{i=1}^n w_i x_i - \theta \quad (1)$$

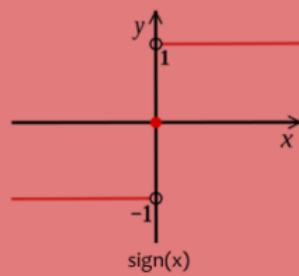
$$y = f(a) = \begin{cases} 0, & \text{if } a \leq 0 \\ 1, & \text{if } a > 0 \end{cases} \quad (2)$$

- The function  $f$  is called the response or activation function:
- in the McCulloch and Pitts neuron  $f$  is simply the step function, so the answer is binary: it is 1 if the weighted sum of the stimuli exceeds the internal threshold; 0 otherwise.
- Other models of artificial neurons predict continuous response functions

# Activation Function

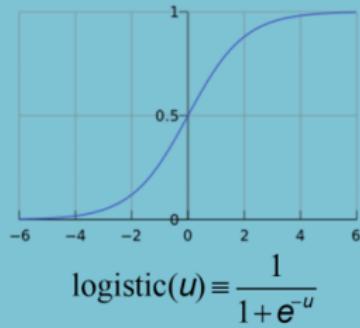
This decision function isn't differentiable:

$$h(\mathbf{x}) = \text{sign}(\boldsymbol{\theta}^T \mathbf{x})$$



Use a differentiable function instead:

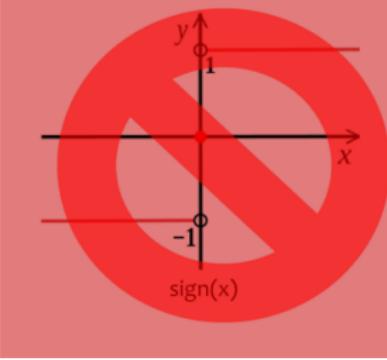
$$p_{\boldsymbol{\theta}}(y = 1 | \mathbf{x}) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^T \mathbf{x})}$$



# Activation Function

This decision function isn't differentiable:

$$h(\mathbf{x}) = \text{sign}(\boldsymbol{\theta}^T \mathbf{x})$$



Use a differentiable function instead:

$$p_{\boldsymbol{\theta}}(y=1|\mathbf{x}) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^T \mathbf{x})}$$

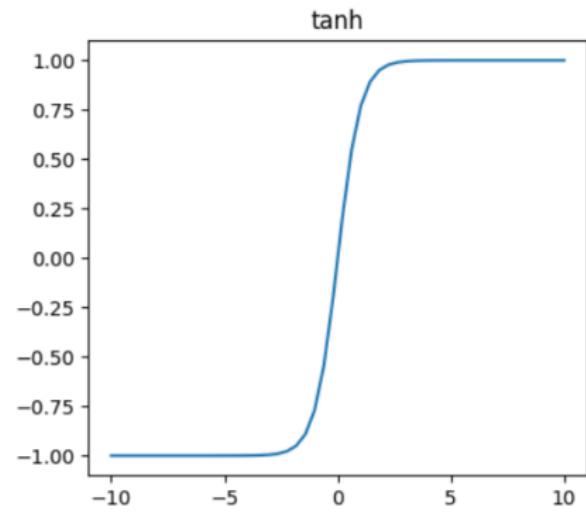
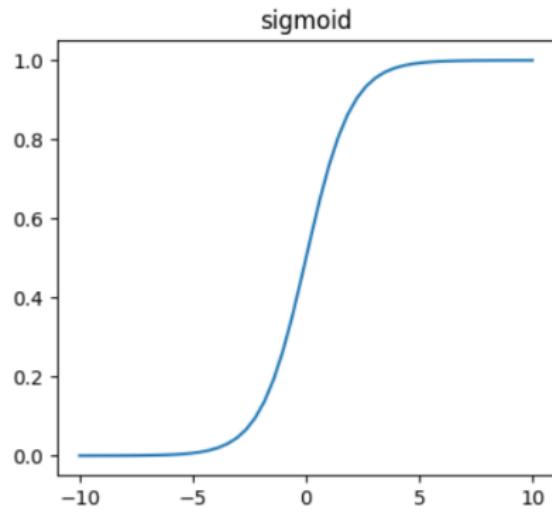


$$\text{logistic}(u) \equiv \frac{1}{1+e^{-u}}$$

# Activation Function

- The non-linear nature of these functions is essential for neural networks to learn from complex data.
- If we only used linear activation functions, no matter how many layers we stacked, the network would behave just like a single-layer perceptron because the composition of linear functions is still a linear function.
- This limits the complexity of tasks the network can learn.
- Non-linear activation functions, on the other hand, enable the network to learn complex patterns and solve intricate problems by adding layers of abstraction.

# Activation Function



# Activation Function

For the derivatives of the activation functions the following rules apply (see notebook chapter-07-01 appendix for details):

## Derivative of Hyperbolic Tangent

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}} \Rightarrow \frac{d}{dx} \tanh x = 1 - (\tanh x)^2 \quad (3)$$

## Derivative of Sigmoid Function

$$\sigma(x) = \left[ \frac{1}{1 + e^{-x}} \right] \Rightarrow \frac{d}{dx} \sigma(x) = \sigma(x) \cdot (1 - \sigma(x)) \quad (4)$$

## Subsection 2

### The Feed-Forward Neural Network

# Neural Network Basic Constituents

More generally, a neural network consists of:

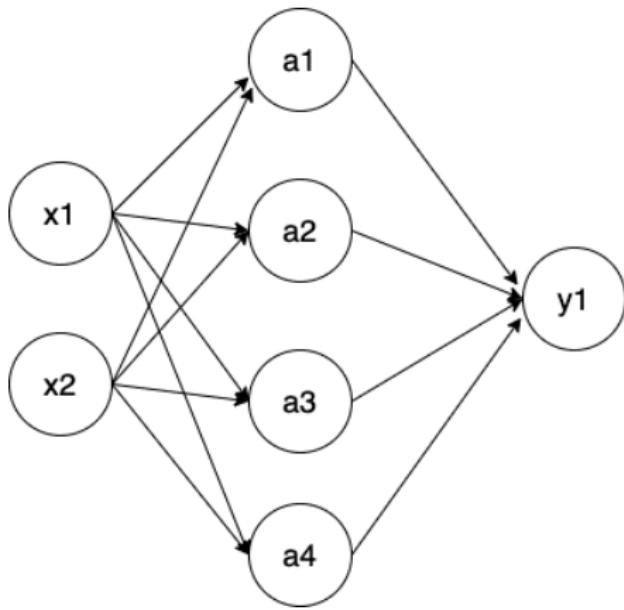
- A set of nodes (neurons), or units connected by links.
- A set of **weights** associated with links.
- A set of thresholds or activation levels.

Neural network design requires:

- 1. The choice of the number and type of units.
- 2. The determination of the morphological structure.
- 3. Coding of training examples, in terms of network inputs and outputs.
- 4. Initialization and training of weights on interconnections, through the set of learning examples.

# A Simple Neural Network

- Feedforward neural networks are what we've primarily been focusing on within this section.
- They are comprised of an input layer, a hidden layer or layers, and an output layer.
- Data is fed into these models through the input layer.

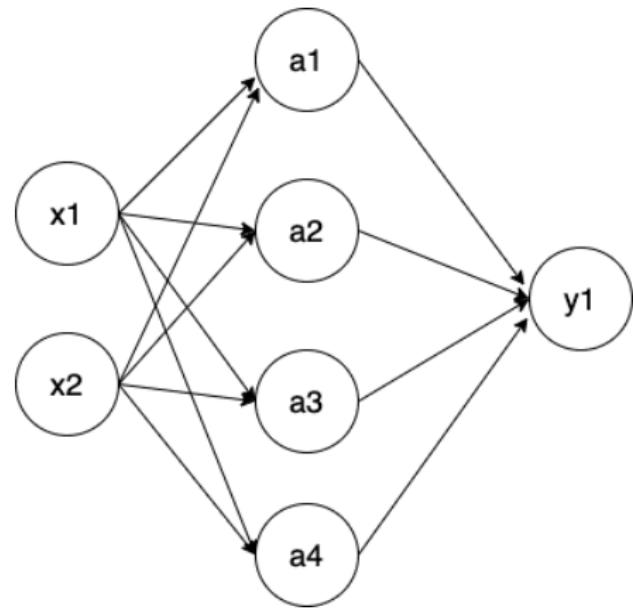


# Implementing a Single Layer NN

- In each hidden unit, take  $a_1$  as example, a **linear operation followed by an activation function,  $f$ , is performed.**
- So given input  $x = (x_1, x_2)$ , **inside node  $a_1$** , we have:

$$z_1 = w_{11}x_1 + w_{12}x_2 + b_1$$

$$a_1 = f(w_{11}x_1 + w_{12}x_2 + b_1) = f(z_1)$$



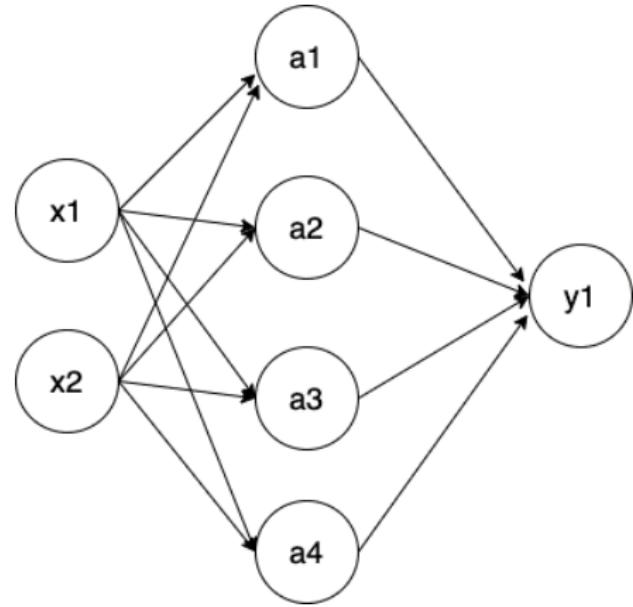
# Implementing a Single Layer NN

- Same for node  $a_2$ , it would have:

$$z_2 = w_{21}x_1 + w_{22}x_2 + b_2$$

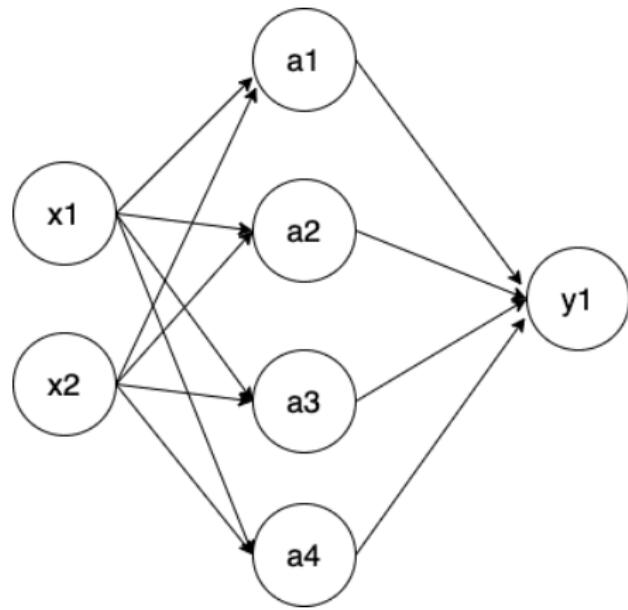
$$a_2 = f(w_{21}x_1 + w_{22}x_2 + b_2) = f(z_2)$$

- And same for  $a_3$  and  $a_4$  and so on



# Implementing a Single Layer NN

- The output is one single value  $y_1$  in  $[0, 1]$ .
- We can think of this simple example as a binary classification task with a prediction of probability



# Implementing a single Layer NN

We can also write in a more compact form the computation of the pre-activation function performed by the hidden layer:

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \Rightarrow Z^{[1]} = W^{[1]} \cdot X + B^{[1]} \quad (5)$$

# Implementing a single Layer NN

Let's assume that the first activation function is the tanh and the output activation function is the *sigmoid*. So the result of the hidden layer is:

$$A^{[1]} = \tanh Z^{[1]}$$

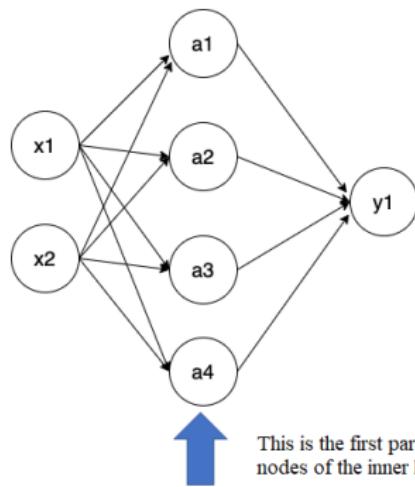
This result is applied to the output node which will perform another linear operation with a different set of weights,  $W^{[2]}$ :

$$Z^{[2]} = W^{[2]} \cdot A^{[1]} + B^{[2]}$$

and the final output will be the result of the application of the output node activation function (the sigmoid) to this value:

$$\hat{y} = \sigma(Z^{[2]}) = A^{[2]}$$

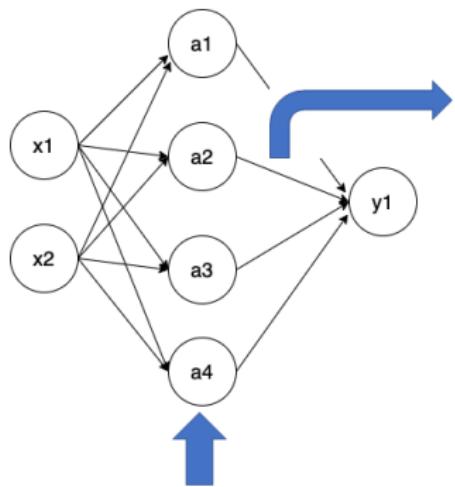
# Implementing a single Layer NN



This is the first part of the computation performed by the nodes of the inner layer ...

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \Rightarrow Z^{[1]} = W^{[1]} \cdot X + B^{[1]}$$

# Implementing a single Layer NN

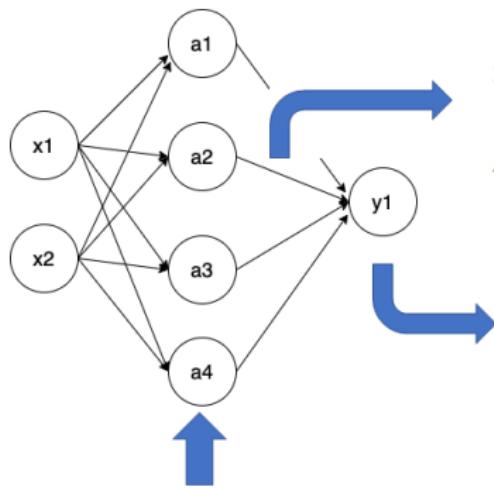


Let's assume that the activation function of the hidden layer is the tanh, so the total result of the hidden layer is

$$A^{[1]} = \tanh Z^{[1]}$$

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \Rightarrow Z^{[1]} = W^{[1]} \cdot X + B^{[1]}$$

# Implementing a single Layer NN



Let's assume that the activation function of the hidden layer is the tanh, so the total result of the hidden layer is

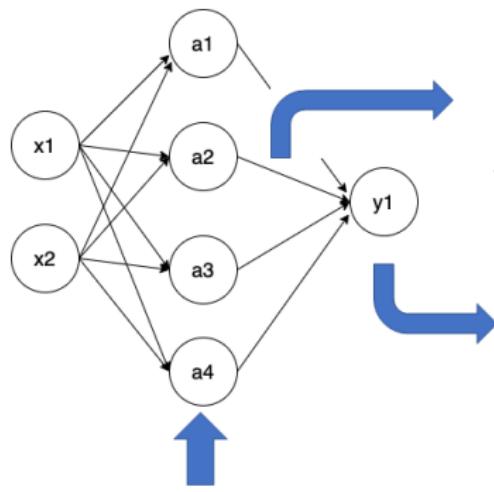
$$A^{[1]} = \tanh Z^{[1]}$$

The previous result is applied to the output node which will perform another linear operation with a **DIFFERENT** set of weights

$$Z^{[2]} = W^{[2]} \cdot A^{[1]} + B^{[2]}$$

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \Rightarrow Z^{[1]} = W^{[1]} \cdot X + B^{[1]}$$

# Implementing a single Layer NN



Let's assume that the activation function of the hidden layer is the tanh, so the total result of the hidden layer is

$$A^{[1]} = \tanh Z^{[1]}$$

The previous result is applied to the output node which will perform another linear operation with a **DIFFERENT** set of weights

$$Z^{[2]} = W^{[2]} \cdot A^{[1]} + B^{[2]}$$

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \Rightarrow Z^{[1]} = W^{[1]} \cdot X + B^{[1]}$$

The Final Output will be the result of the application of the output node activation function (sigmoid):

$$\hat{y} = \sigma(Z^{[2]}) = A^{[2]}$$

# Neural Network Basic Constituents

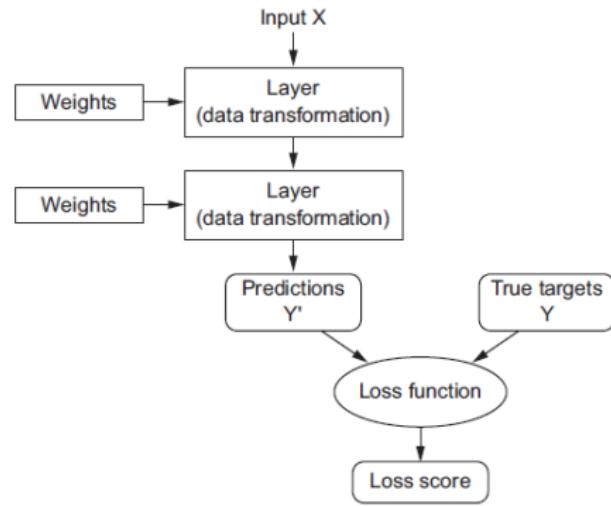
$$\begin{aligned}\hat{y} &= A^{[2]} = \sigma(Z^{[2]}) = \sigma(W^{[2]} \cdot A^{[1]} + B^{[2]}) \\ &= \sigma(W^{[2]} \cdot \tanh Z^{[1]} + B^{[2]}) \\ &= \sigma\left[W^{[2]} \cdot \tanh\left(W^{[1]} \cdot X + B^{[1]}\right) + B^{[2]}\right]\end{aligned}$$

---

- The specification of what a layer does to its input data is stored in the layer's weights, which in essence are a bunch of numbers.
- In technical terms, we could say that the transformation implemented by a layer is parameterized by its weights (Weights are also sometimes called the parameters of a layer.)
- In this context, **learning means finding a set of values for the weights of all layers in a network, such that the network will correctly map example inputs to their associated targets.**

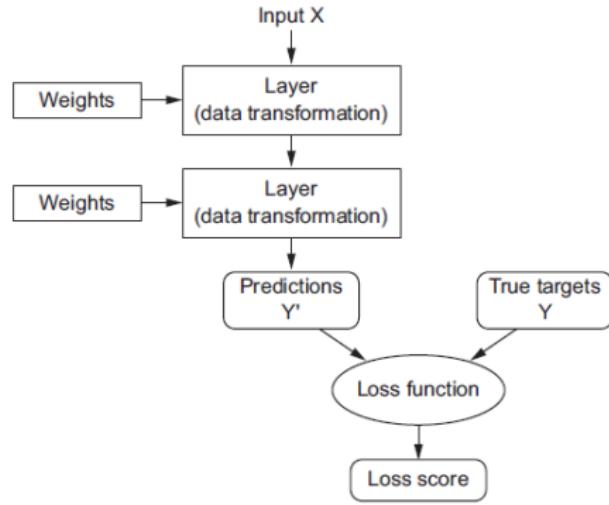
# Loss Function

- To control the output of a neural network, you need to be able to measure how far this output is from what you expected.
- This is the job of the **loss function** of the network, also called the objective function.



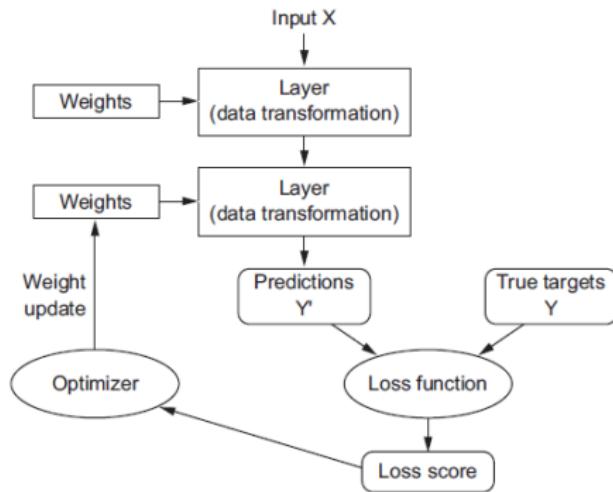
# Loss Function

- The loss function takes the predictions of the network and the true target (what you wanted the network to output) and **computes a distance score, capturing how well the network has done on this specific example**

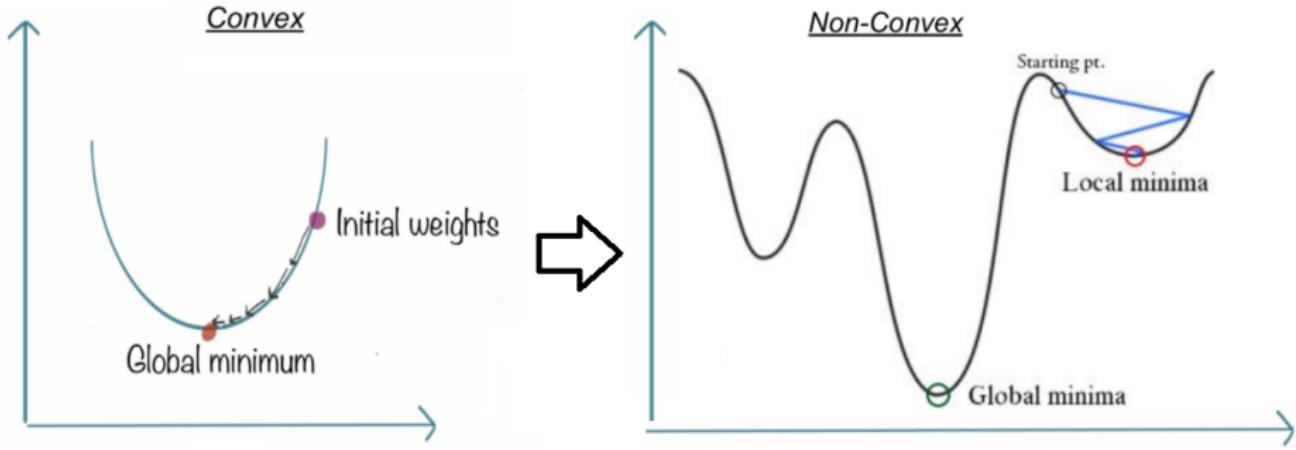


# Backpropagation

- The fundamental trick in deep learning is to use this score as a feedback signal to adjust the value of the weights a little, in a direction that will lower the loss score for the current example.
- This adjustment is the job of the optimizer, which implements what is called the Backpropagation algorithm: the central algorithm in deep learning.



# Loss Function



# Weights Initialization

- Our neural network has 1 hidden layer and 2 layers in total (hidden layer + output layer), so there are 4 weight matrices to initialize ( $W^{[1]}, b^{[1]}$  and  $W^{[2]}, b^{[2]}$ ).
  - Notice that the weights are initialized relatively small so that the gradients would be higher thus learning faster in the beginning phase.
- 

```
def init_weights(n_input, n_hidden, n_output):  
    params = {}  
    params['W1'] = np.random.randn(n_hidden, n_input) * 0.01  
    params['b1'] = np.zeros((n_hidden, 1))  
    params['W2'] = np.random.randn(n_output, n_hidden) * 0.01  
    params['b2'] = np.zeros((n_output, 1))  
  
    return params
```

---

# Weights Initialization

- Our neural network has 1 hidden layer and 2 layers in total (hidden layer + output layer), so there are 4 weight matrices to initialize ( $W^{[1]}, b^{[1]}$  and  $W^{[2]}, b^{[2]}$ ).
  - Notice that the weights are initialized relatively small so that the gradients would be higher thus learning faster in the beginning phase.
- 

```
params = init_weights(20, 10, 1)

print('W1 shape', params['W1'].shape)
print('b1 shape', params['b1'].shape)
print('W2 shape', params['W2'].shape)
print('b2 shape', params['b2'].shape)
```

---

# Forward Propagation

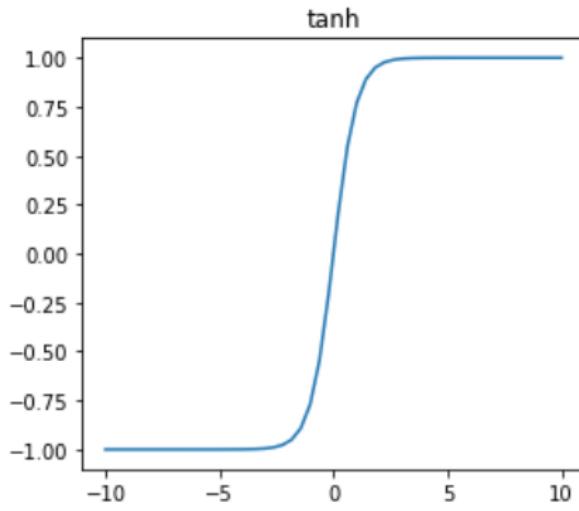
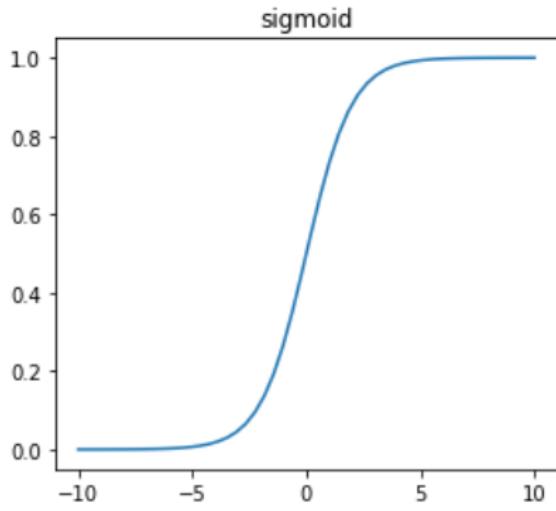
$$\begin{aligned}\hat{y} &= A^{[2]} = \sigma(Z^{[2]}) = \sigma(W^{[2]} \cdot A^{[1]} + B^{[2]}) \\ &= \sigma(W^{[2]} \cdot \tanh Z^{[1]} + B^{[2]}) \\ &= \sigma\left[W^{[2]} \cdot \tanh\left(W^{[1]} \cdot X + B^{[1]}\right) + B^{[2]}\right]\end{aligned}$$

---

```
def forward(X, params):
    W1, b1, W2, b2 = params['W1'], params['b1'], params['W2'], params['b2']
    A0 = X
    cache = {}
    Z1 = np.dot(W1, A0) + b1
    A1 = tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)
    cache['Z1'] = Z1
    cache['A1'] = A1
    cache['Z2'] = Z2
    cache['A2'] = A2
    return cache
```

# Activation Functions

Function tanh and sigmoid looks as below. Notice that the only difference of these functions is the scale of y.



# Logistic Loss Function

Since we have a binary classification problem, we can assume a Logistic Loss Function (see the problem of logistic regression)

$$L(y, \hat{y}) = \begin{cases} -\log \hat{y} & \text{when } y = 1 \\ -\log(1 - \hat{y}) & \text{when } y = 0 \end{cases} \quad (6)$$

$$L(y, \hat{y}) = -[y \log \hat{y} + (1 - y) \log (1 - \hat{y})]$$

Where  $\hat{y}$  is our prediction ranging in  $[0, 1]$  and  $y$  is the true value.

# Logistic Loss Function

$$L(y, \hat{y}) = -[y \log \hat{y} + (1 - y) \log (1 - \hat{y})]$$

---

```
def loss(Y, Y_hat):
    """
    Y: vector of true value
    Y_hat: vector of predicted value
    """
    assert Y.shape[0] == 1
    assert Y.shape == Y_hat.shape
    m = Y.shape[1]
    s = Y * np.log(Y_hat) + (1 - Y) * np.log(1 - Y_hat)
    loss = -np.sum(s) / m
    return loss
```

---

# Delta Rule

- Given a generic actual value  $y$ , we want to minimize the loss  $L$ , and the technique we are going to apply here is gradient descent;
- basically what we need to do is to apply derivative to our variables and move them slightly down to the optimum.
- Here we have 2 variables,  $W$  and  $b$ , and for this example, the update formula of them would be:

$$W_{new} = W_{old} - \frac{\partial L}{\partial W} \Rightarrow \Delta W = -\frac{\partial L}{\partial W}$$
$$b_{new} = b_{old} - \frac{\partial L}{\partial b} \Rightarrow \Delta b = -\frac{\partial L}{\partial b}$$

# Delta Rule

- The delta rule algorithm works by computing the gradient of the loss function with respect to each weight.
- Remember that

$$\begin{aligned}\hat{y} &= A^{[2]} = \sigma(Z^{[2]}) = \sigma(W^{[2]} \cdot A^{[1]} + B^{[2]}) \\ &= \sigma(W^{[2]} \cdot \tanh Z^{[1]} + B^{[2]}) \\ &= \sigma\left[W^{[2]} \cdot \tanh\left(W^{[1]} \cdot X + B^{[1]}\right) + B^{[2]}\right]\end{aligned}$$

As you can see  $\hat{y}$  depends on both  $W^{[1]}$  and  $W^{[2]}$ . The specification of what a layer does to its input data is stored in the layer's weights. Remember once again that **learning means finding a set of values for the weights of all layers in a network, such that the network will correctly map example inputs to their associated targets**

# Delta Rule

The delta rule algorithm works by computing the gradient of the loss function with respect to each weight. In order to get the derivative of our targets, chain rules would be applied:

$$\Delta W^{[1]} = -\frac{\partial L}{\partial W^{[1]}} \quad (7)$$

$$\Delta W^{[2]} = -\frac{\partial L}{\partial W^{[2]}} \quad (8)$$

$$\frac{\partial L}{\partial W^{[i]}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial Z^{[2]}} \frac{\partial Z^{[2]}}{\partial W^{[i]}} \quad (9)$$

where  $i = 1, 2$

# Delta Rule

Given the loss function  $L$  we defined above, we can easily compute the first partial derivative with respect to  $\hat{y}$ :

$$\begin{aligned} L(y, \hat{y}) &= -[y \log \hat{y} + (1 - y) \log (1 - \hat{y})] \\ &\Downarrow \\ \frac{\partial L}{\partial \hat{y}} &= -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}} = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})} \end{aligned}$$

Remember that in our notation we have:

$$\hat{y} = \sigma(Z^{[2]}) = A^{[2]}$$

# Delta Rule

So we can write for the two first partial derivatives of our chain:

$$\frac{\partial L}{\partial \hat{y}} = \frac{A^{[2]} - y}{A^{[2]} (1 - A^{[2]})}$$

$$\begin{aligned}\frac{\partial \hat{y}}{\partial Z^{[2]}} &= \frac{\partial \sigma(Z^{[2]})}{\partial Z^{[2]}} \\ &= \sigma(Z^{[2]}) \cdot [1 - \sigma(Z^{[2]})] \\ &= \hat{y}(1 - \hat{y}) = A^{[2]}(1 - A^{[2]})\end{aligned}$$

# Delta Rule

putting it all together

$$\begin{aligned}\frac{\partial L}{\partial W^{[i]}} &= \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial W^{[i]}} \\ &= \frac{A^{[2]} - y}{A^{[2]}(1 - A^{[2]})} \cdot A^{[2]}(1 - A^{[2]}) \cdot \frac{\partial Z^{[2]}}{\partial W^{[i]}} \\ &= (A^{[2]} - y) \cdot \frac{\partial Z^{[2]}}{\partial W^{[i]}}\end{aligned}$$

where  $i = 1, 2$

# Delta Rule

Now we have only to compute the partial derivatives of the pre-activation function  $Z^{[2]}$  with respect to  $W^{[1]}$  and  $W^{[2]}$ .

Remember that

$$Z^{[2]} = W^{[2]} \cdot A^{[1]} + B^{[2]} = W^{[2]} \cdot \tanh\left(W^{[1]} \cdot X + B^{[1]}\right) + B^{[2]}$$

where

$$A^{[1]} = \tanh\left(W^{[1]} \cdot X + B^{[1]}\right) = \tanh\left(Z^{[1]}\right)$$

# Delta Rule

We have

$$\frac{\partial Z^{[2]}}{\partial W^{[2]}} = A^{[1]}$$

$$\frac{\partial Z^{[2]}}{\partial W^{[1]}} = W^{[2]} \cdot \frac{\partial \tanh(Z^{[1]})}{\partial Z^{[1]}} \cdot \frac{\partial Z^{[1]}}{\partial W^{[1]}}$$

$$\begin{cases} \frac{\partial \tanh(Z^{[1]})}{\partial Z^{[1]}} = 1 - [\tanh(Z^{[1]})]^2 = 1 - (A^{[1]})^2 \\ \frac{\partial Z^{[1]}}{\partial W^{[1]}} = X \end{cases} \quad (10)$$

$$\frac{\partial Z^{[2]}}{\partial W^{[1]}} = W^{[2]} \cdot X \cdot [1 - (A^{[1]})^2]$$

# Delta Rule

Now we have all the elements to calculate the derivative of the loss function with respect to the weights

$$\frac{\partial L}{\partial W^{[1]}} = (A^{[2]} - y) \cdot \frac{\partial Z^{[2]}}{\partial W^{[1]}} = (A^{[2]} - y) \cdot W^{[2]} \cdot X \cdot \left[ 1 - (A^{[1]})^2 \right]$$

$$\frac{\partial L}{\partial W^{[2]}} = (A^{[2]} - y) \cdot \frac{\partial Z^{[2]}}{\partial W^{[2]}} = (A^{[2]} - y) \cdot A^{[1]}$$

# Delta Rule

In the python code we put also:

$$\Delta^{[2]} = A^{[2]} - y$$

$$\Delta^{[1]} = \Delta^{[2]} \cdot W^{[2]} \cdot (1 - A^{[1]2})$$

So we finally have

$$\Delta W^{[2]} = -\frac{\partial L}{\partial W^{[2]}} = -\Delta^{[2]} \cdot A^{[1]}$$

$$\Delta W^{[1]} = -\frac{\partial L}{\partial W^{[1]}} = -\Delta^{[1]} \cdot X$$

# Weights Update

---

```
def backward(params, cache, X, Y):
    m = X.shape[1]
    W1 = params['W1']
    W2 = params['W2']
    A1 = cache['A1']
    A2 = cache['A2']
    DL2 = A2 - Y
    dW2 = (1 / m) * np.dot(DL2, A1.T)
    db2 = (1 / m) * np.sum(DL2, axis=1, keepdims=True)
    DL1 = np.multiply(np.dot(W2.T, DL2), 1 - np.power(A1, 2))
    dW1 = (1 / m) * np.dot(DL1, X.T)
    db1 = (1 / m) * np.sum(DL1, axis=1, keepdims=True)
    grads = {"dW1": dW1,
              "db1": db1,
              "dW2": dW2,
              "db2": db2}

    return grads
```

# Batch Training

- In actual training processes, a batch is trained instead of 1 at a time. The change applied in the formula is trivial, we just need to replace the single vector  $x$  with a matrix  $X$  with size  $n \times m$ , where  $n$  is number of features and  $m$  is the the batch size, samples are stacked column wise, and the following result matrix are applied likewise.
- Also the loss function is the same as logistic regression, but for batch training, we'll take the average loss for all training samples.
- The same it's true for all the other calculation, this explain the presence of the factor  $1/m$ ;

## Subsection 3

### Keras

# Keras

Keras is a popular open-source deep learning library that is written in Python. It is designed to be user-friendly, modular, and extensible, which makes it a great tool for beginners and experts alike. Here are some key points about Keras:

- User-friendly API: Keras provides a high-level API that makes it easy to build and train deep learning models. It supports both CPU and GPU computations, and it runs seamlessly on different platforms.
- Modular architecture: Keras is built on a modular architecture, which means that it provides a set of building blocks that you can combine to create complex deep learning models. This allows you to experiment with different architectures and see what works best for your problem.

# Keras

- Supports multiple backends: Keras supports multiple backends, including TensorFlow, CNTK, and Theano. This allows you to choose the backend that works best for your needs.
- Pre-built models and layers: Keras provides a set of pre-built models and layers that you can use to jumpstart your deep learning projects. These models and layers are optimized for different tasks, such as image recognition, natural language processing, and time series analysis.
- Easy customization: Keras allows you to customize your models and layers easily. You can add or remove layers, change the hyperparameters, and fine-tune the models to improve their performance.

# Keras

- Built-in utilities: Keras provides a set of built-in utilities that make it easy to preprocess your data, visualize your models, and monitor your training progress. These utilities can save you a lot of time and effort when working on deep learning projects.
- Large community: Keras has a large and active community of users and contributors, who share their knowledge and experience through forums, blogs, and tutorials. This means that you can find resources and help easily when you run into issues or have questions.
- Integration with other libraries: Keras integrates seamlessly with other Python libraries, such as NumPy, Pandas, and Scikit-learn. This allows you to use these libraries together with Keras to preprocess your data, analyze your results, and build end-to-end deep learning pipelines.

# Introduction to keras: a practical problem

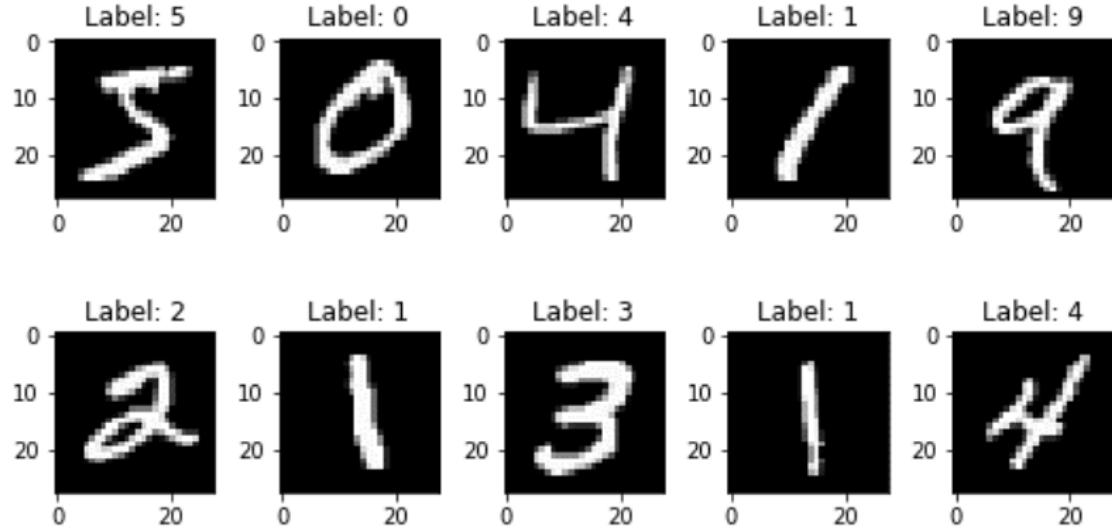
- The problem of automatic recognition of handwritten numbers, also known as digit recognition, is a classic problem in the field of pattern recognition and computer vision. It involves the task of automatically identifying and classifying handwritten digits into their corresponding numerical values.
- The challenge of digit recognition arises due to the variability in handwriting styles and the different ways in which people write the same digit. This variation can be caused by factors such as different writing instruments, writing speed, and writing pressure. Additionally, digits can be written in different sizes, orientations, and positions, further adding to the complexity of the problem.

# Introduction to keras: a practical problem

- The problem we are trying to solve here is to classify grayscale images of handwritten digits (28 pixels by 28 pixels), into their 10 categories (0 to 9).
- The dataset we will use is the MNIST dataset, a classic dataset in the machine learning community, which has been around for almost as long as the field itself and has been very intensively studied.
- It's a set of 60,000 training images, plus 10,000 test images, assembled by the National Institute of Standards and Technology (the NIST in MNIST) in the 1980s.

# Introduction to keras: a practical problem

- As we have said, in the MNIST dataset each digit is stored in a grayscale image with a size of 28x28 pixels.
- In the following you can see the first 10 digits from the training set:



# Introduction to keras: Loading the dataset

- Keras provides seven different datasets, which can be loaded in using Keras directly.
- These include image datasets as well as a house price and a movie review datasets.
- The MNIST dataset comes pre-loaded in Keras, in the form of a set of four Numpy arrays

---

```
import keras

from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

---

# Introduction to keras

## The typical Keras workflows

- Define your training data: input tensor and target tensor
- Define a network of layers(or model ) that maps input to our targets.
- Configure the learning process by choosing a loss function, an optimizer, and some metrics to monitor.
- Iterate your training data by calling the fit() method of your model.

# Introduction to keras: Layers

- This is the building block of neural networks which are stacked or combined together to form a neural network model.
- It is a data-preprocessing module that takes one or more input tensors and outputs one or more tensors.
- These layers together contain the network's knowledge.
- Different layers are made for different tensor formats and data processing.

# Creating a model with the sequential API

- The easiest way of creating a model in Keras is by using the sequential API, which lets you stack one layer after the other.
  - The problem with the sequential API is that it doesn't allow models to have multiple inputs or outputs, which are needed for some problems.
  - Nevertheless, the sequential API is a perfect choice for most problems.
- 

```
from keras import models
from keras import layers

network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

---

# Introduction to keras

Let's go through this code line by line:

- The first line creates a Sequential model.
  - This is the simplest kind of Keras model, for neural networks that are just composed of a single stack of layers, connected sequentially.
  - This is called the **sequential** API.
- 

```
from keras import models
from keras import layers

network = models.Sequential()
```

---

# Introduction to keras

Next, we build the first layer and add it to the model.

- It is **Dense** hidden layer with 512 neurons.
- It will use the ReLU activation function.
- Each Dense layer manages its own weight matrix, containing all the connection weights between the neurons and their inputs.
- It also manages a vector of bias terms (one per neuron).
- When it receives some input data, it computes

$$\phi \left( Z^{[1]} = W^{[1]} \cdot X + B^{[1]} \right), \quad \phi(z) = \text{ReLU}(z)$$

---

```
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
```

---

# Introduction to keras

Finally, we add a Dense output layer with 10 neurons (one per class).

- Using a 10-way "softmax" layer means that it will return an array of 10 probability scores (summing to 1).
  - Each score will be the probability that the current digit image belongs to one of our 10 digit classes.
- 

```
network.add(layers.Dense(10, activation='softmax'))
```

---

# Introduction to keras

- The model's `summary()` method displays all the model's layers, including each layer's name (which is automatically generated unless you set it when creating the layer), its output shape (None means the batch size can be anything), and its number of parameters.
- 

```
network.summary()
```

---

# Introduction to keras

- The summary ends with the total number of parameters, including trainable and non-trainable parameters.
- Here we only have trainable parameters.

Model: "sequential\_1"

Layer (type)	Output Shape	Param #
=====		
dense_2 (Dense)	(None, 512)	401920
dense_3 (Dense)	(None, 10)	5130
=====		
Total params: 407,050		
Trainable params: 407,050		
Non-trainable params: 0		
=====		

# Compile a Model

- Before we can start training our model we need to configure the learning process.
- For this, we need to specify an optimizer, a loss function and optionally some metrics like accuracy.
- The **loss function** is a measure on how good our model is at achieving the given objective.
- An **optimizer** is used to minimize the loss(objective) function by updating the weights using the gradients.

# Loss Function

- Choosing the right Loss Function for the problem is very important, the neural network can take any shortcut to minimize the loss.
- So, if the objective doesn't fully correlate with success for the task at hand, your network will end up doing things you may not have wanted.

# Loss Function

For common problems like Classification, Regression and Sequence prediction, they are simple guidelines to choose a loss function.

For:

- Two- Class classification you can choose binary cross-entropy
- Multi-Class Classification you can choose Categorical Cross-entropy.
- Regression Problem you can choose Mean-Squared Error

# Activation Function

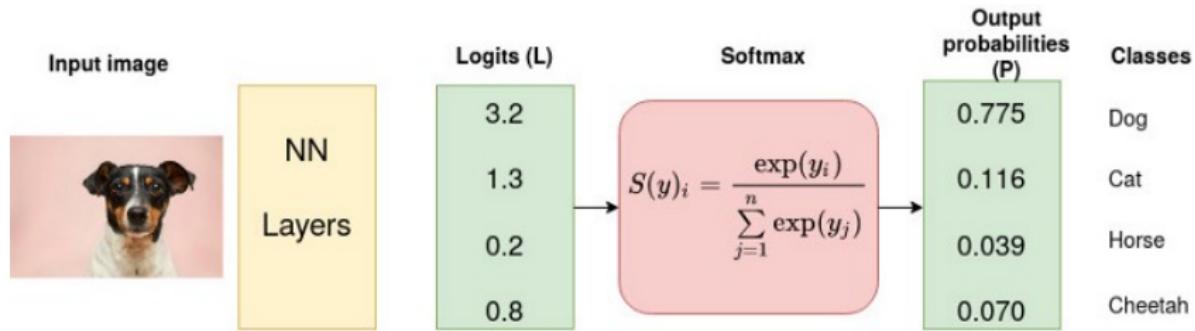
## Softmax Activation Function

- Softmax is an activation function that scales numbers/logits into probabilities.
- The output of a Softmax is a vector (say  $v$ ) with probabilities of each possible outcome.
- The probabilities in vector  $v$  sums to one for all possible outcomes or classes
- It is often used as the last activation function of a neural network to normalize the output of a network to a probability distribution over predicted output classes
- You can think of softmax function as a sort of generalization to multiple dimension of the logistic function

# Activation Function

**Softmax Activation Function.** Mathematically, softmax is defined as

$$S(y)_i = \frac{\exp y_i}{\sum_{j=1}^n \exp y_j} \quad (11)$$



# Loss Function

## Categorical Cross-Entropy

- Remember the logistic Loss Function:

$$L(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}) = -\sum_{j=1}^2 y_j \log \hat{y}_j$$

- Generalizing this result to the n-class classification problem we have

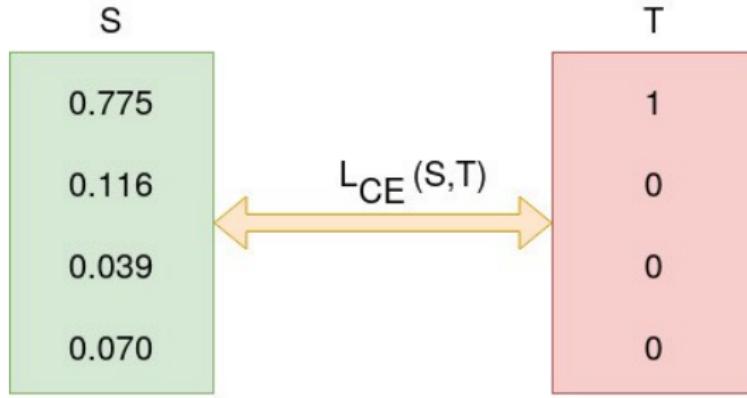
$$L = -\sum_{j=1}^{N_C} y_j \log \hat{y}_j \quad (12)$$

This last equation define the so called *cross-entropy*

# Loss Function

## Categorical Cross-Entropy

- In the previous example, Softmax converts logits into probabilities.
- The purpose of the Cross-Entropy is to take the output probabilities ( $P$ ) and measure the distance from the truth values



# Loss Function

## Categorical Cross-Entropy

- The categorical cross-entropy is computed as follows

$$\begin{aligned}L_{CE} &= - \sum_{i=1} T_i \log(S_i) \\&= - [1 \log_2(0.775) + 0 \log_2(0.126) + 0 \log_2(0.039) + 0 \log_2(0.070)] \\&= - \log_2(0.775) \\&= 0.3677\end{aligned}$$

# Optimizer

## RMSProp

- Gradient descent is an optimization algorithm that follows the negative gradient of an objective function in order to locate the minimum of the function.
- A limitation of gradient descent is that it uses the same step size (learning rate) for each input variable.
- AdaGrad is an extension of the gradient descent optimization algorithm that allows the step size in each dimension used by the optimization algorithm to be automatically adapted based on the gradients seen for the variable (partial derivatives) over the course of the search.
- A limitation of AdaGrad is that it can result in a very small step size for each parameter by the end of the search that can slow the progress of the search down too much and may mean not locating the optima.
- Root Mean Squared Propagation, or RMSProp, is an extension of gradient descent and the AdaGrad version of gradient descent that uses a decaying average of partial gradients in the adaptation of the step size for each parameter.
- The use of a decaying moving average allows the algorithm to forget early gradients and focus on the most recently observed partial gradients seen during the progress of the search, overcoming the limitation of AdaGrad.

# Compile the model

So, to make our network ready for training, we need to pick three things, as part of "compilation" step:

- A loss function: this is how the network will be able to measure how good a job it is doing on its training data, and thus how it will be able to steer itself in the right direction.
  - An optimizer: this is the mechanism through which the network will update itself based on the data it sees and its loss function.
  - Metrics to monitor during training and testing. Here we will only care about accuracy (the fraction of the images that were correctly classified).
- 

```
network.compile(optimizer='rmsprop',
                 loss='categorical_crossentropy',
                 metrics=['accuracy'])
```

---

# Introduction to keras: Training

- Before training, we will preprocess our data by reshaping it into the shape that the network expects, and scaling it so that all values are in the '[0, 1]' interval.
  - Previously, our training images for instance were stored in an array of shape '(60000, 28, 28)' of type 'uint8' with values in the '[0, 255]' interval.
  - We transform it into a 'float32' array of shape '(60000, 28 \* 28)' with values between 0 and 1.
- 

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

---

# Introduction to keras: Training

We also need to categorically encode the labels:

---

```
from keras.utils.np_utils import to_categorical  
  
train_labels = to_categorical(train_labels)  
test_labels = to_categorical(test_labels)
```

---

# Introduction to keras: Training

## What is an epoch?

- An epoch is a term used in machine learning and indicates the number of passes of the entire training dataset the machine learning algorithm has completed. Datasets are usually grouped into batches (especially when the amount of data is very large). Some people use the term iteration loosely and refer to putting one batch through the model as an iteration.
- If the batch size is the whole training dataset then the number of epochs is the number of iterations. For practical reasons, this is usually not the case. Many models are created with more than one epoch. The general relation where dataset size is  $d$ , number of epochs is  $e$ , number of iterations is  $i$ , and batch size is  $b$  would be  $d \cdot e = i \cdot b$ .
- Determining how many epochs a model should run to train is based on many parameters related to both the data itself and the goal of the model, and while there have been efforts to turn this process into an algorithm, often a deep understanding of the data itself is indispensable.

# Introduction to keras: Training

We are now ready to train our network, which in Keras is done via a call to the 'fit' method of the network: we "fit" the model to its training data.

---

```
history = network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

---

# Visualizing the training process

- We can visualize our training and testing accuracy and loss for each epoch so we can get intuition about the performance of our model.
  - The accuracy and loss over epochs are saved in the history variable we got whilst training and we will use Matplotlib to visualize this data.
- 

```
# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['mae'])
plt.plot(history.history['val_mae'])
plt.title('model mae')
plt.ylabel('mae')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

---



# Other Neural Network Architectures

- Up to this point, we've described a specific neural network architecture (feed forward NN) where values flow forward linearly through a network, and gradients flow linearly backwards through a network.
- However, this is just the tip of the iceberg when it comes to the field of neural networks.
- While feed forward neural networks have been incredibly successful in a wide range of applications, many other types of neural network architectures exist that can be used to solve different types of problems.
- In particular in this lesson we are going to describe Convolutional Neural Network and Recurrent Neural Network

# Convolutional Neural Network

- Convolutional Neural Networks (CNNs) are specifically designed for processing spatial data, such as images.
- Unlike feed-forward networks, CNNs use special convolutional layers to scan and identify local patterns within the input.
- Imagine a grid sliding across an image, identifying patterns.
- This makes them more efficient for image recognition, object detection, and other computer vision tasks, where spatial information is crucial.
- They are also used for sequential tasks, such as time-series applications.

# Recurrent Neural Network

- Recurrent Neural Networks (RNNs) differ from feed-forward neural networks as they have a built-in memory, allowing them to process sequences of data.
- This makes RNNs well-suited for tasks like natural language processing and time series prediction.
- They can learn patterns in sequences by connecting the output from one time step to the input of the next, remembering previous information (the recurrence in the namesake).

## Subsection 1

### Convolutional Neural Network

# Convolutional Neural Network

- A convolutional neural network, or CNN for short, is a type of artificial neural network that is commonly used for image recognition and classification tasks.
- It is inspired by the organization of the visual cortex in animals and is designed to process visual data in a way that is similar to how the human brain processes visual information.
- The main idea behind a CNN is to use a series of **convolutional layers** to automatically learn features from the input image, such as edges, lines, and shapes.
- These features are then combined and fed into a series of fully connected layers that perform the final classification task.

# Convolutional Layer

- A **convolutional layer** is a key building block in convolutional neural networks (CNNs) used for image recognition and computer vision tasks.
- It performs a mathematical operation called convolution on the input image to extract features that are useful for the given task.

1	0	1
0	1	0
1	0	1

Filter / Kernel

1 <small><math>\times 1</math></small>	1 <small><math>\times 0</math></small>	1 <small><math>\times 1</math></small>	0	0
0 <small><math>\times 0</math></small>	1 <small><math>\times 1</math></small>	1 <small><math>\times 0</math></small>	1	0
0 <small><math>\times 1</math></small>	0 <small><math>\times 0</math></small>	1 <small><math>\times 1</math></small>	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved Feature

# What is Convolution

- The convolution operation involves sliding a small matrix of numbers called a **kernel** or **filter** over the input image.
- The kernel is typically much smaller than the input image and contains learnable weights that are optimized during the training process.

1	0	1
0	1	0
1	0	1

Filter / Kernel

1 <small><math>\times 1</math></small>	1 <small><math>\times 0</math></small>	1 <small><math>\times 1</math></small>	0	0
0 <small><math>\times 0</math></small>	1 <small><math>\times 1</math></small>	1 <small><math>\times 0</math></small>	1	0
0 <small><math>\times 1</math></small>	0 <small><math>\times 0</math></small>	1 <small><math>\times 1</math></small>	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved Feature

# What is a Convolution

Convolutional layers typically have multiple kernels that are applied to the input image to extract different features. These features may include edges, textures, and patterns that are relevant to the given image recognition task.



Original



Sharpen



Edge Detect



“Strong” Edge Detect

# What is Convolution

- An easy way to understand this is if you were a detective and you came across a large image or a picture in dark, how will you identify the image?
- You will use your flashlight and scan across the entire image. This is exactly what we do in convolutional layer.
- Kernel K, which is a feature detector is equivalent of the flashlight on image I, and we are trying to detect feature to help us identify or classify the image.

1	0	1
0	1	0
1	0	1

Filter / Kernel

1 $\times 1$	1 $\times 0$	1 $\times 1$	0	0
0 $\times 0$	1 $\times 1$	1 $\times 0$	1	0
0 $\times 1$	0 $\times 0$	1 $\times 1$	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved Feature

# What is Convolution

- As the kernel slides over the input image, it computes a dot product between its weights and the pixel values of the corresponding region of the image.
- The output of this dot product is a single number that is used to create a new output image or feature map.
- Just like the weights in a fully connected layer, the kernel weights are learned during training, and adjusted after each training iteration through backpropagation.

1	0	1
0	1	0
1	0	1

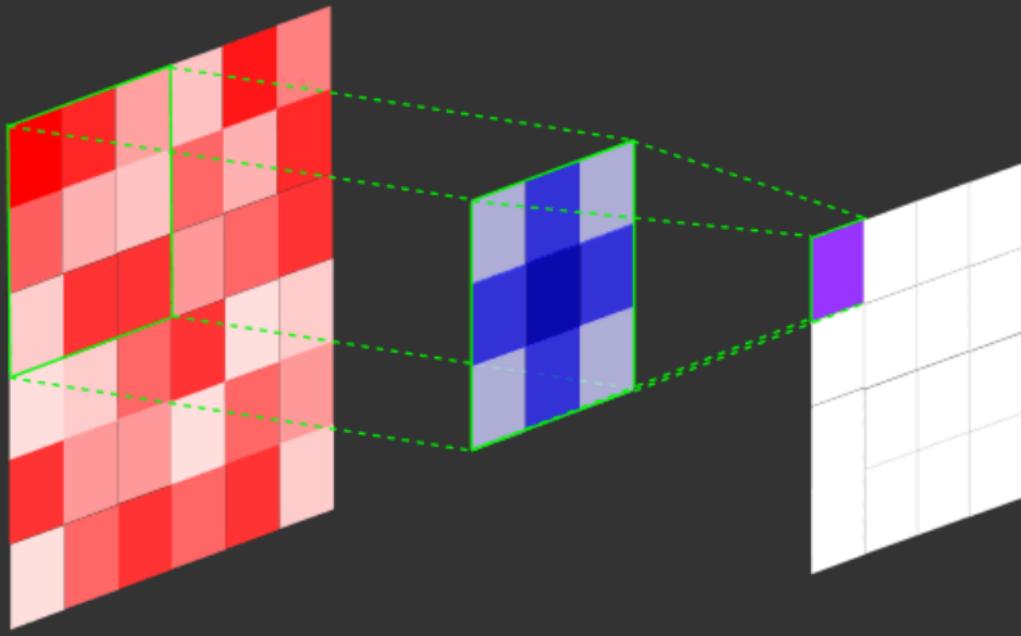
Filter / Kernel

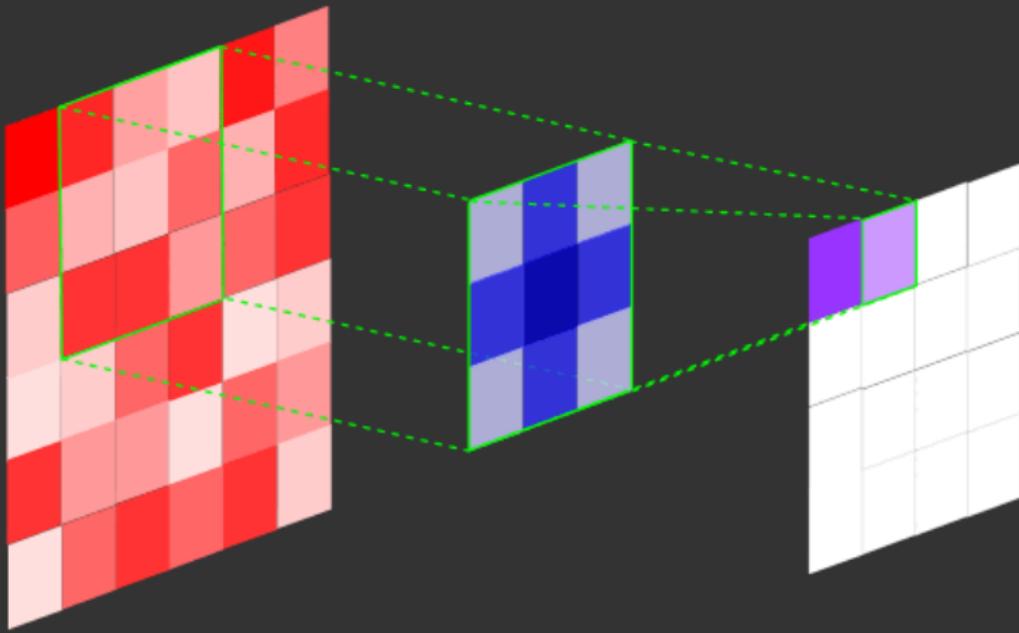
1 $\times 1$	1 $\times 0$	1 $\times 1$	0	0
0 $\times 0$	1 $\times 1$	1 $\times 0$	1	0
0 $\times 1$	0 $\times 0$	1 $\times 1$	1	1
0	0	1	1	0
0	1	1	0	0

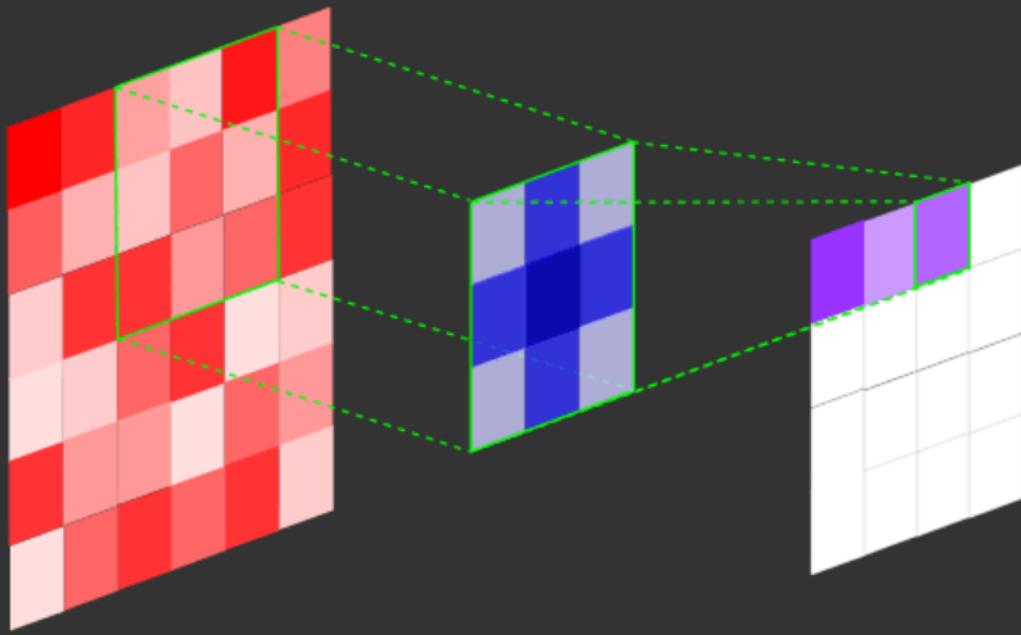
Image

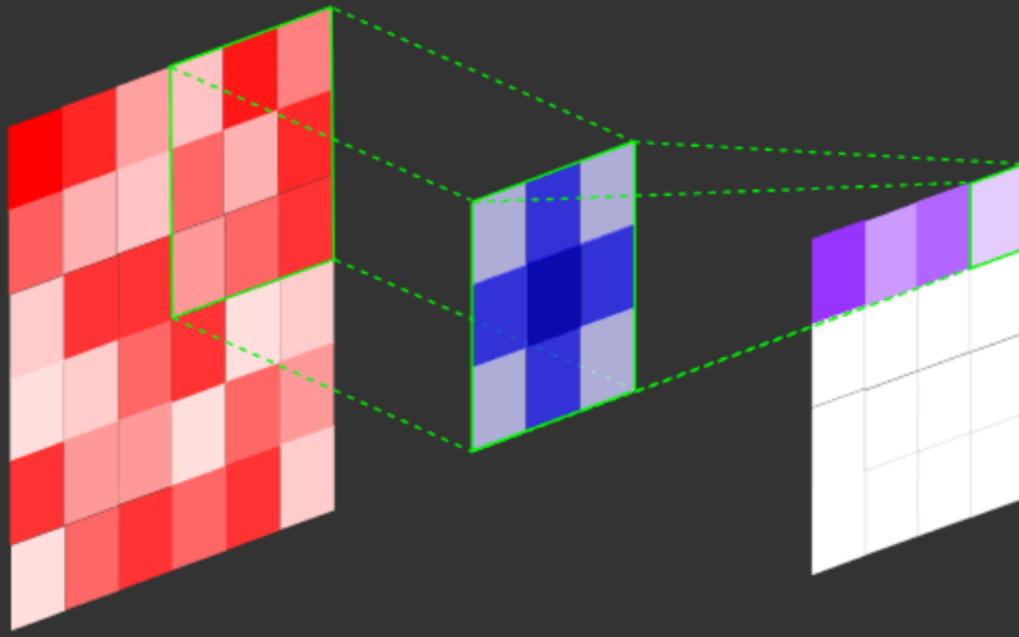
4		

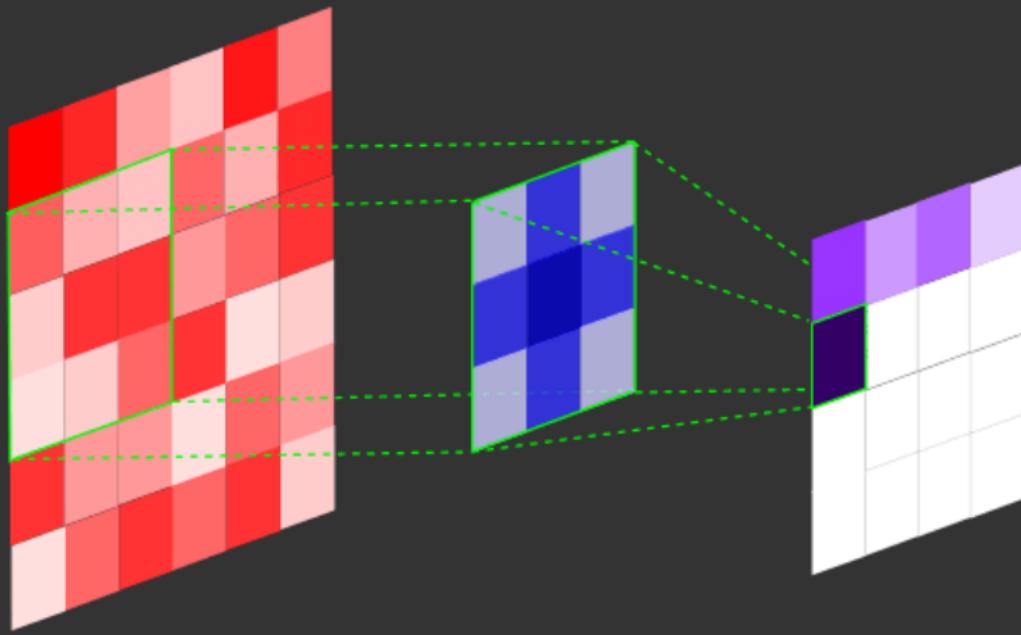
Convolved Feature

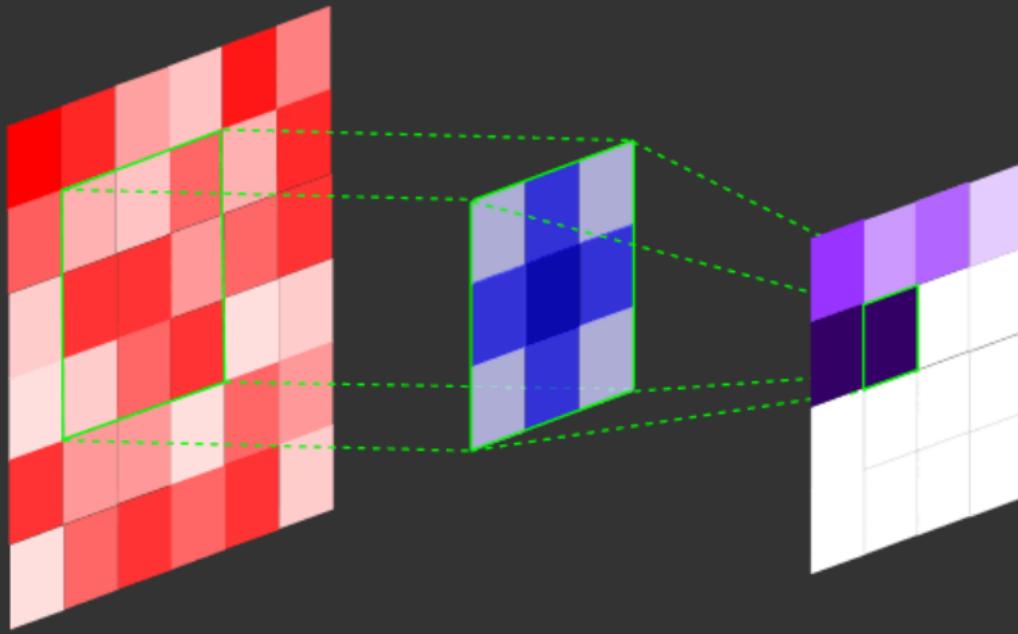


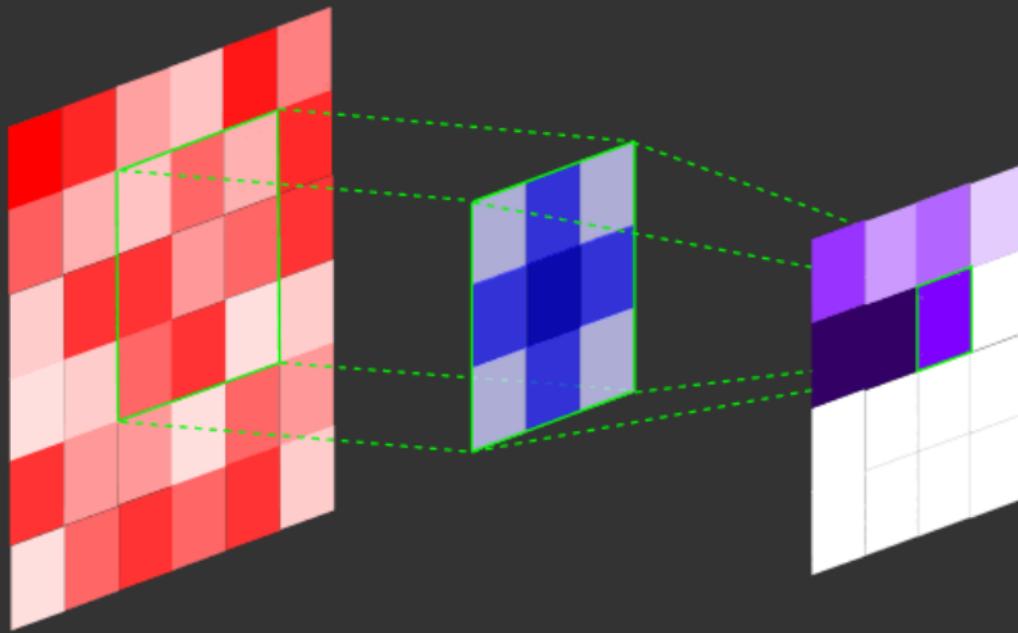


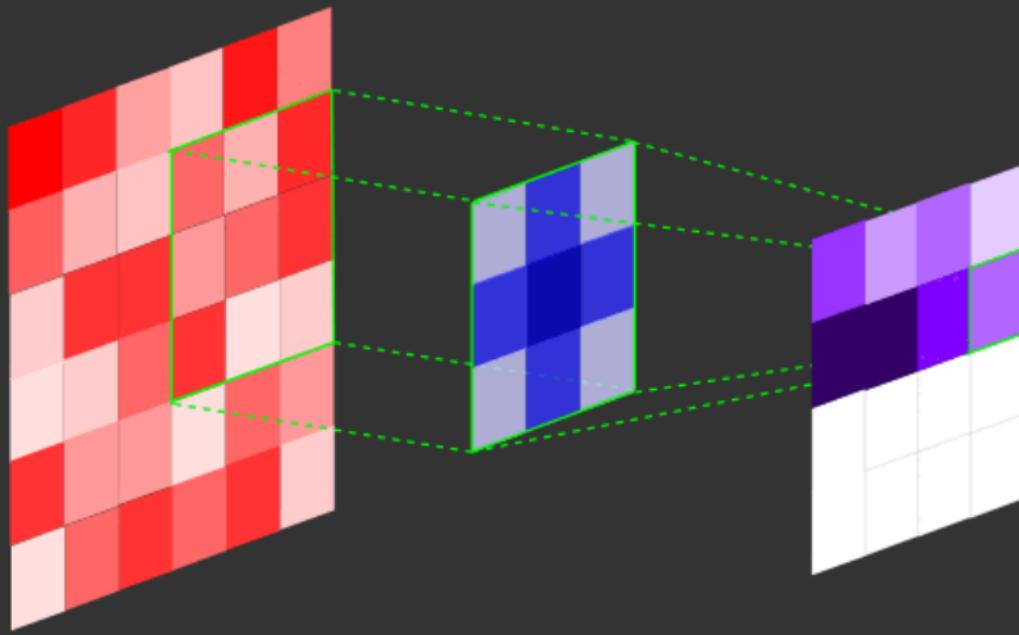


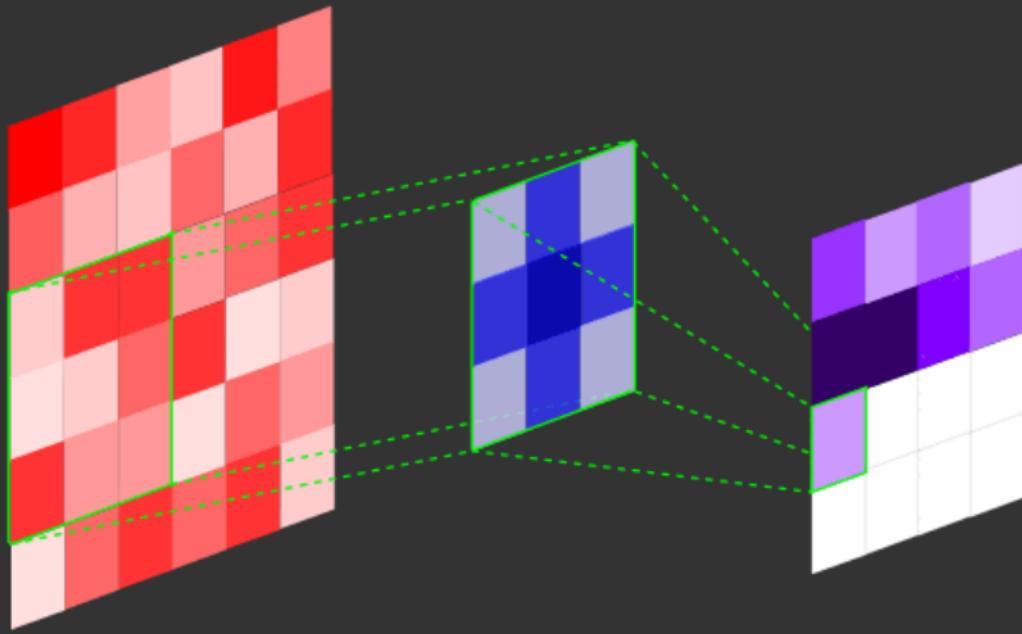


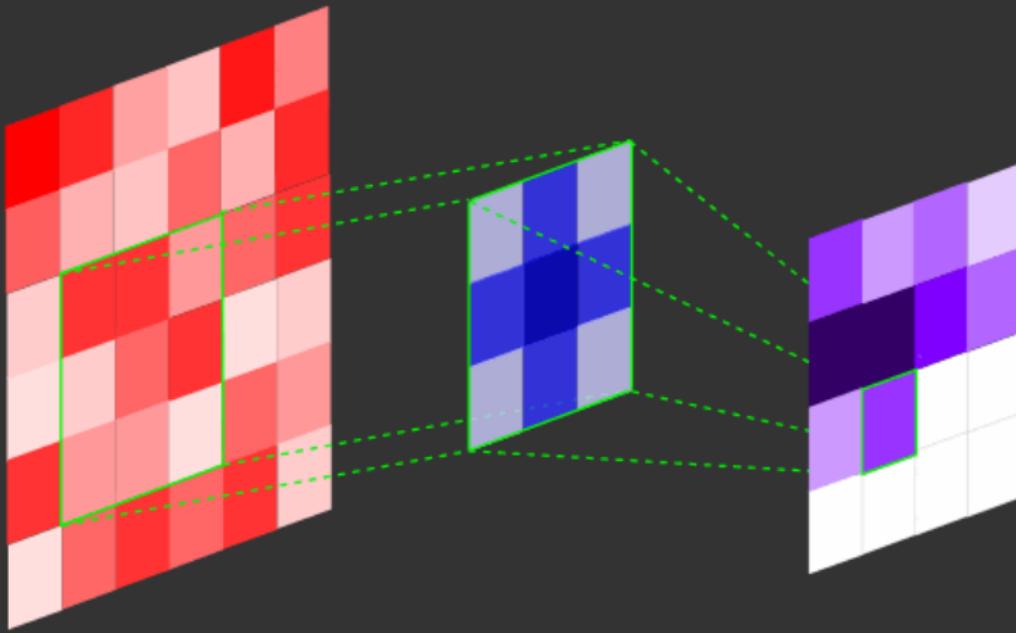


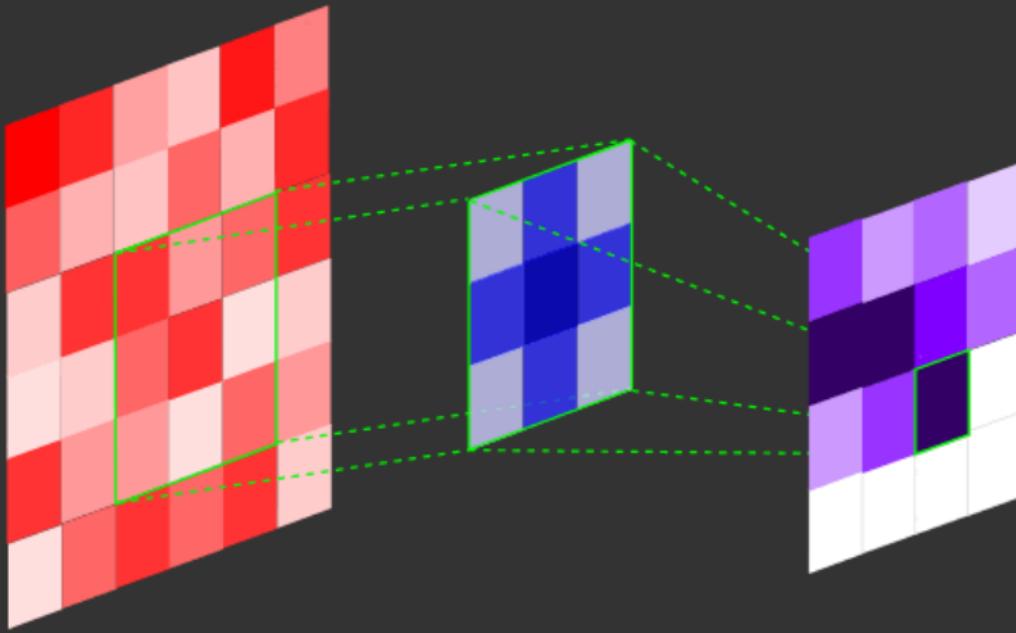


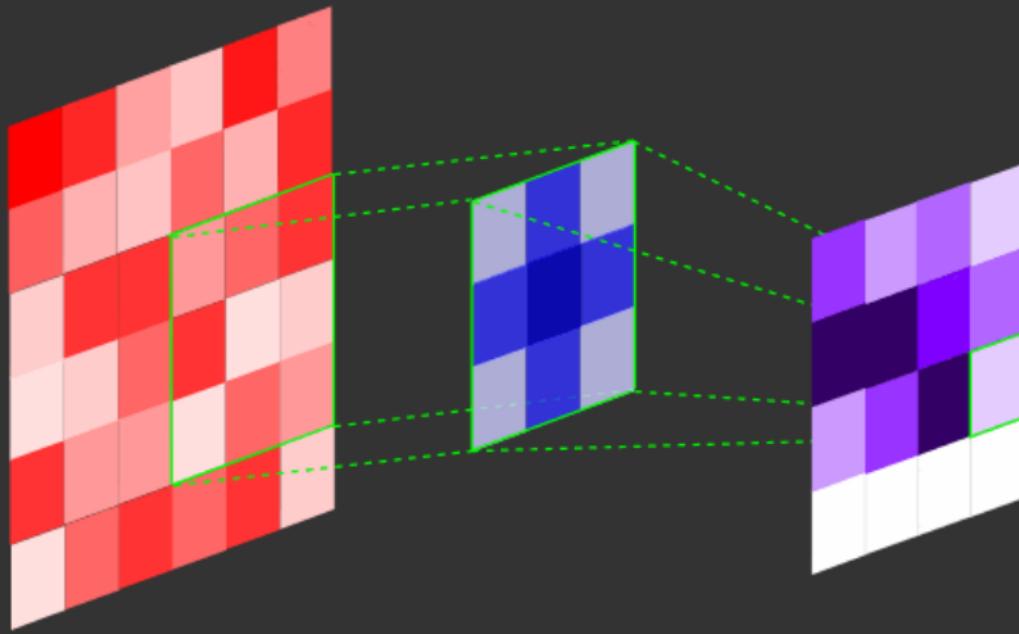


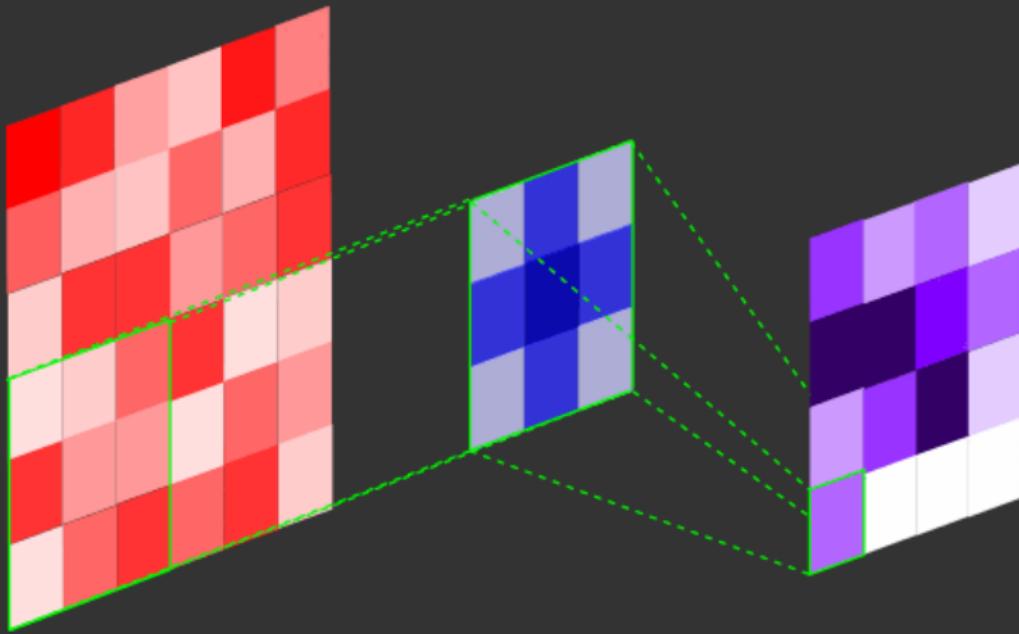


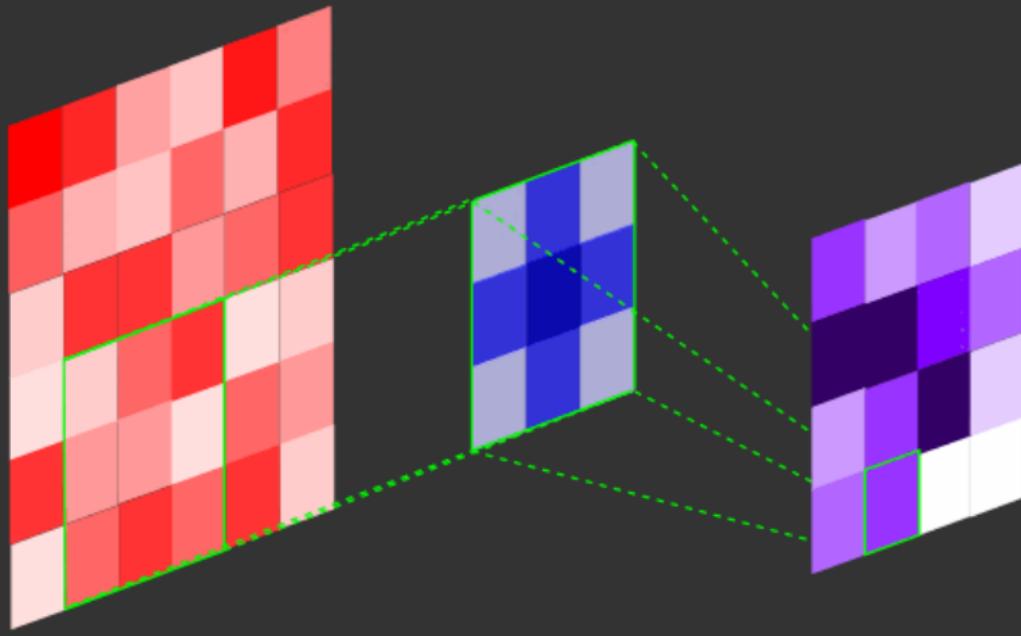


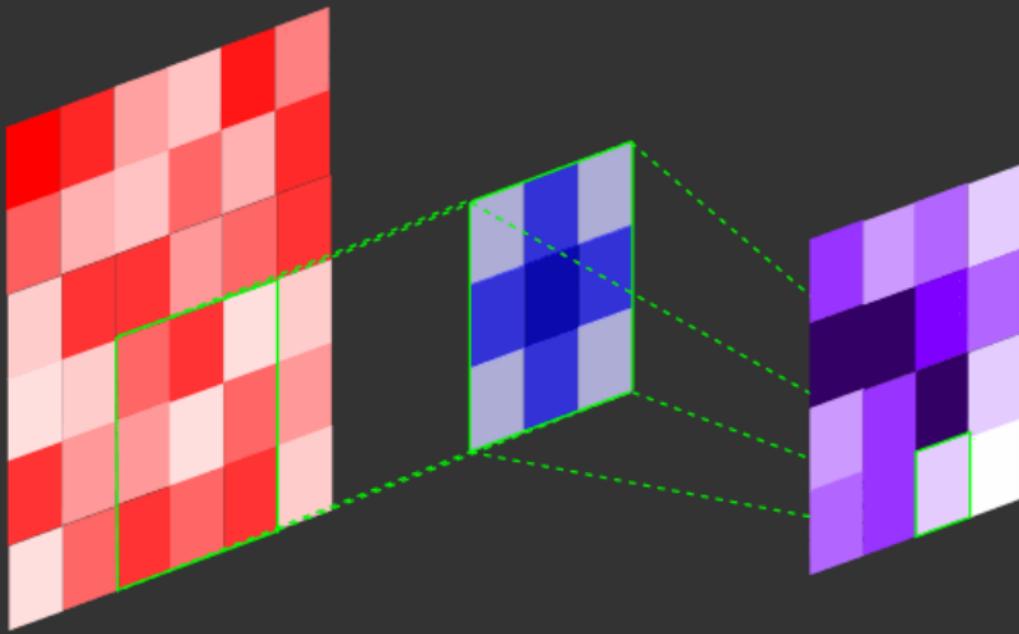


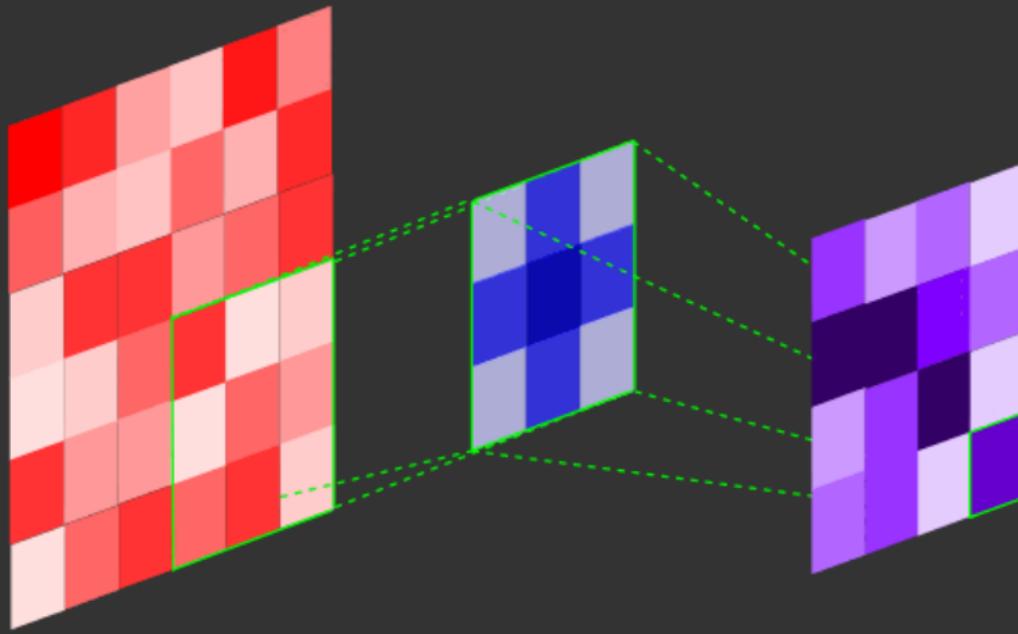


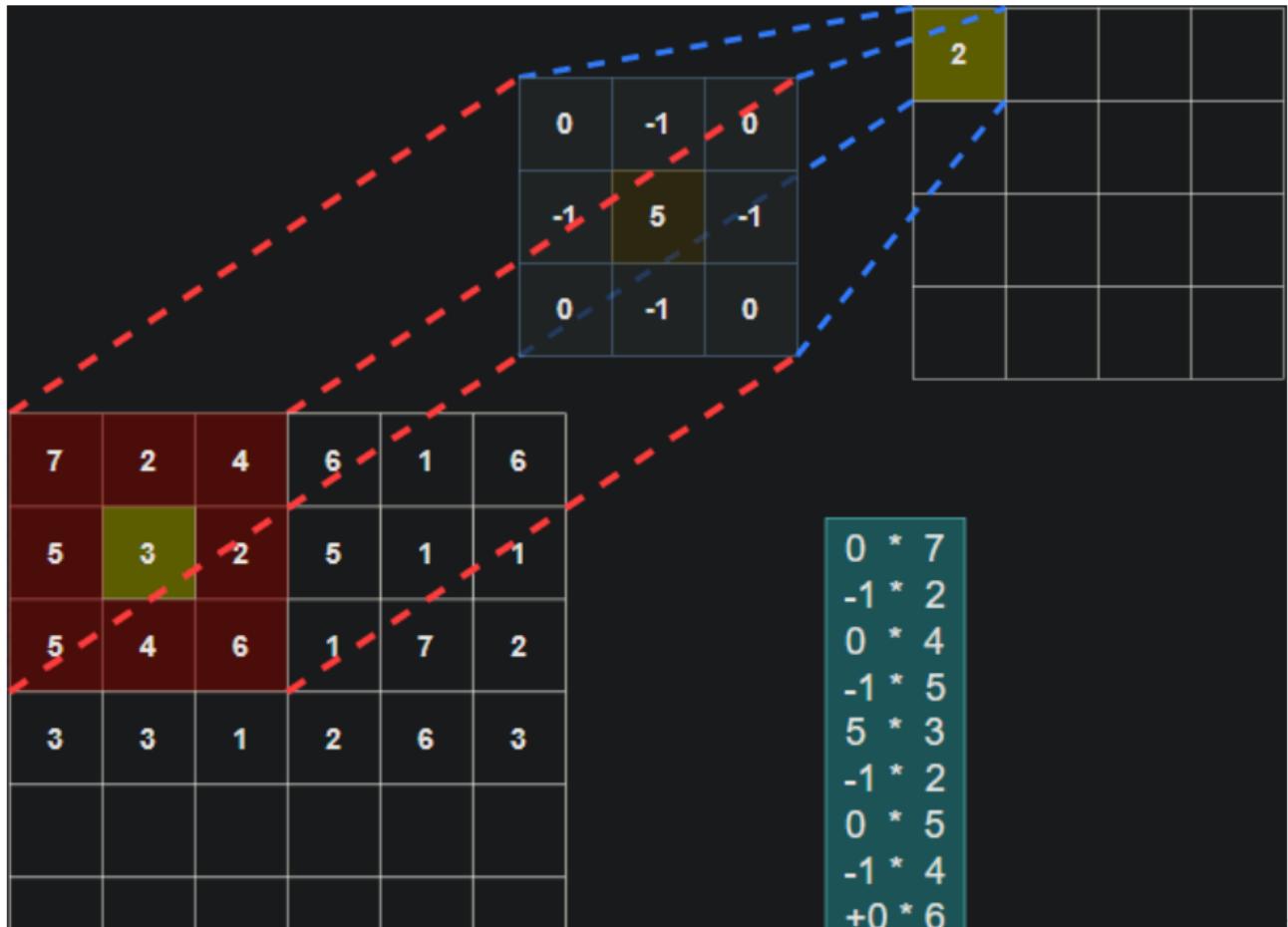


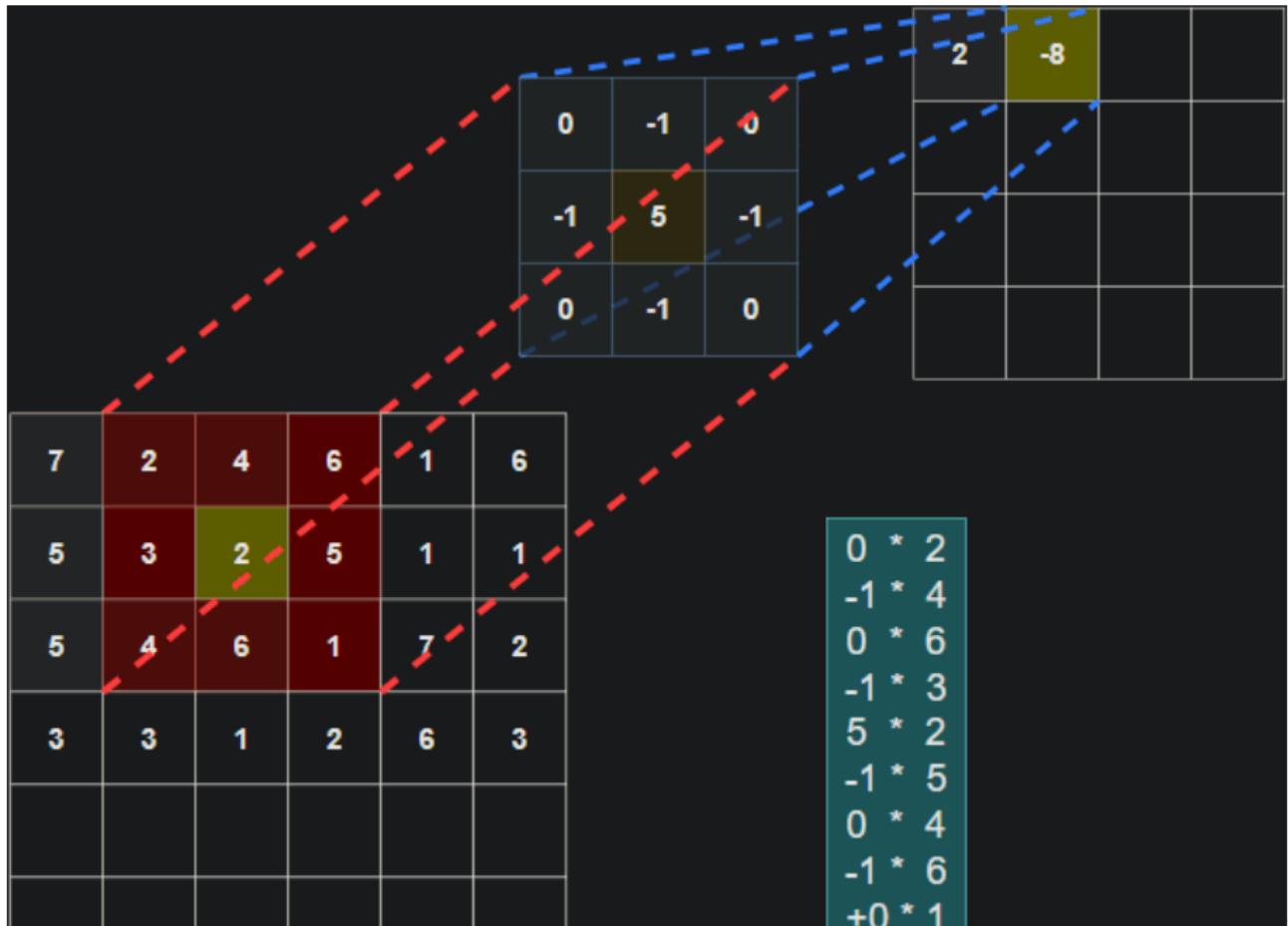


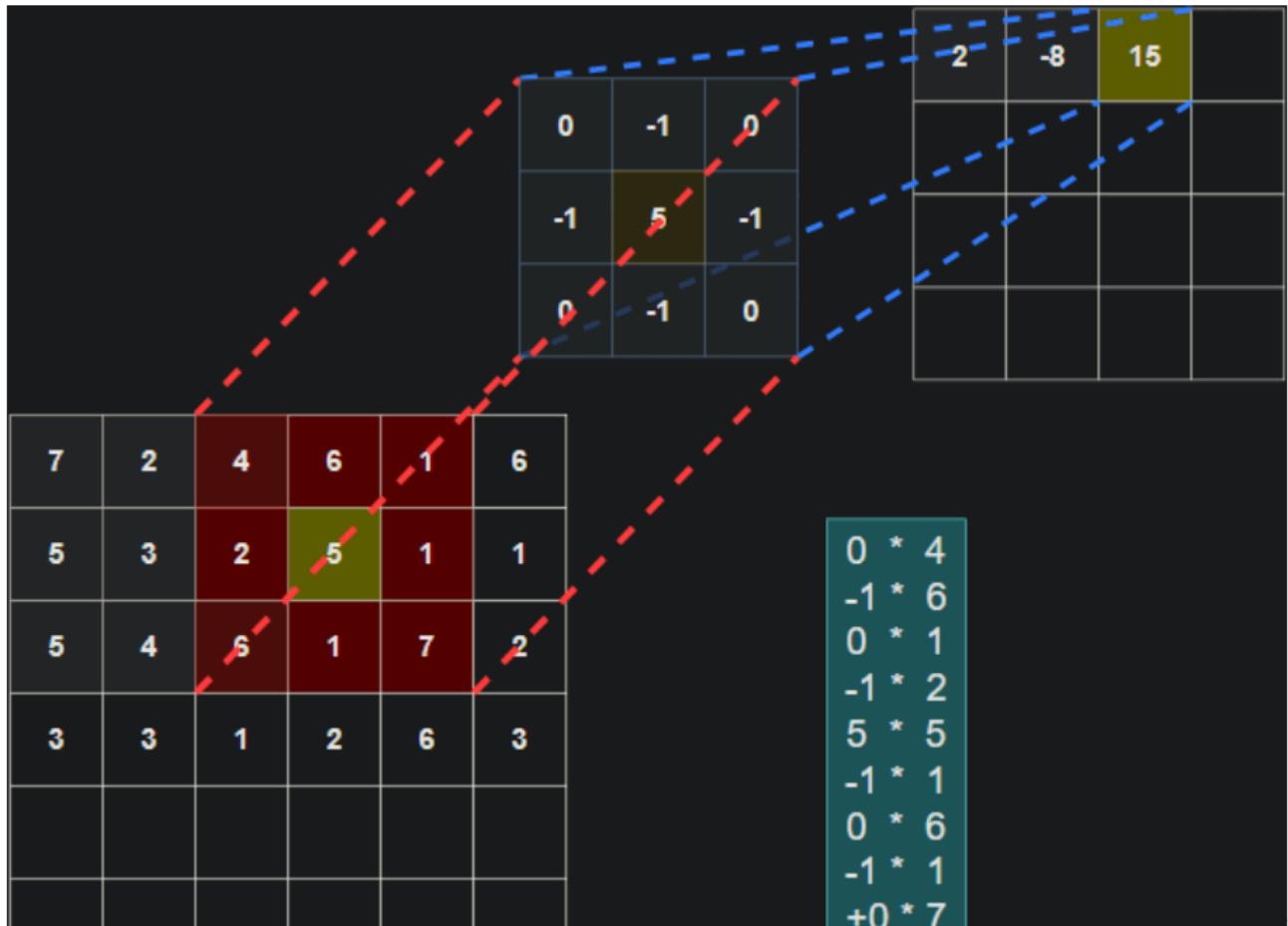


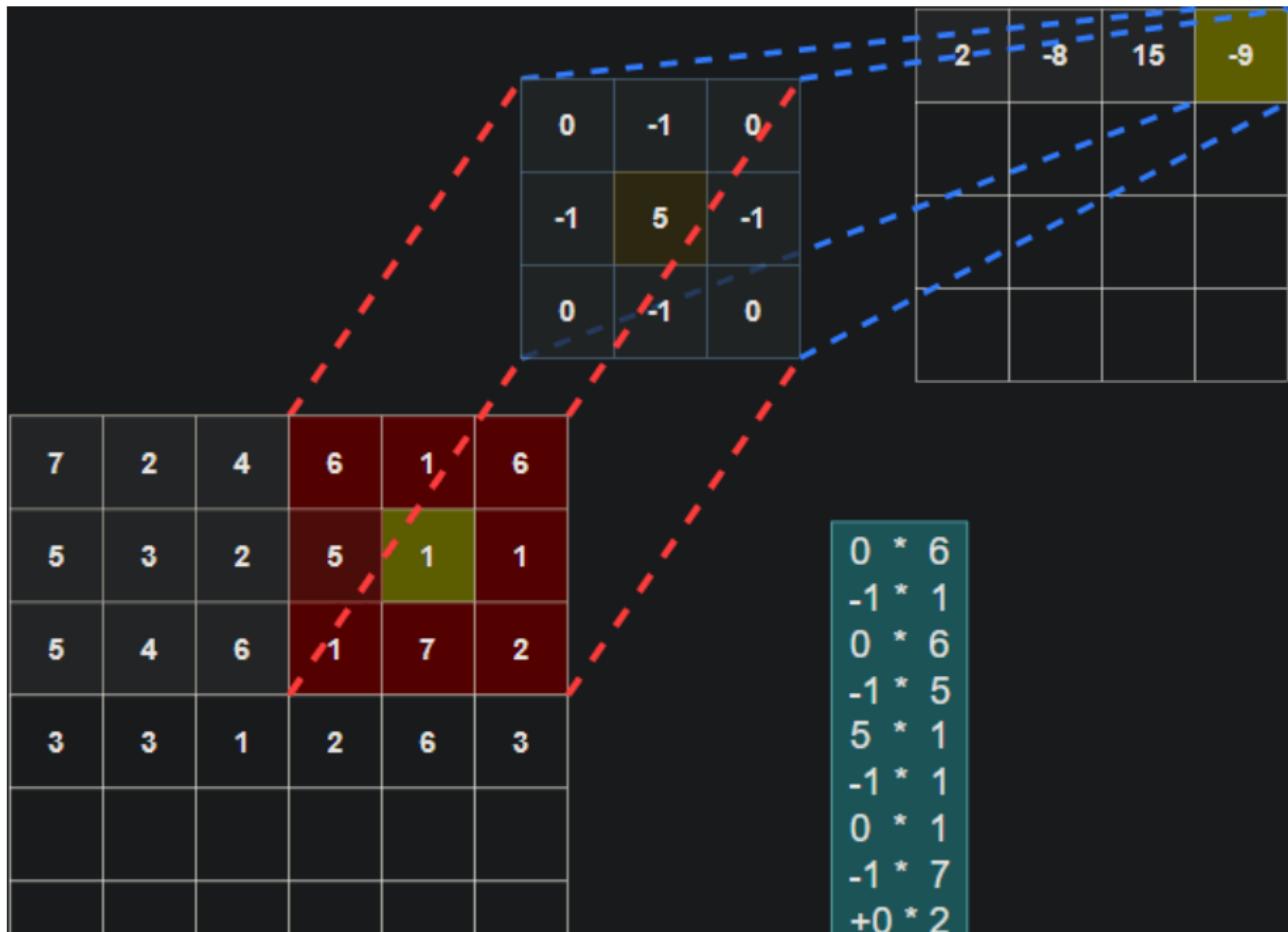












# Producing Features Map

Convolutional layers typically have multiple kernels that are applied to the input image to extract different features. These features may include edges, textures, and patterns that are relevant to the given image recognition task.



Original



Sharpen



Edge Detect



“Strong” Edge Detect

# Producing Features Map

Let's take an image  $x$ , which is a 2D array of pixels with different color channels(Red,Green and Blue-RGB). If we have a **feature detector or kernel  $w$**  then the output we get after applying the previous mathematical operation is called a **feature map**

$$s[t] = (x \star w)[t] = \sum_{a=-\infty}^{a=\infty} x[a]w[a+t]$$

The diagram illustrates the convolution operation. It shows three components: 'Feature map' (the result), 'Input' (the original image), and 'kernel' (the feature detector). Arrows point from each component to its corresponding term in the mathematical expression.

# Producing Features Map

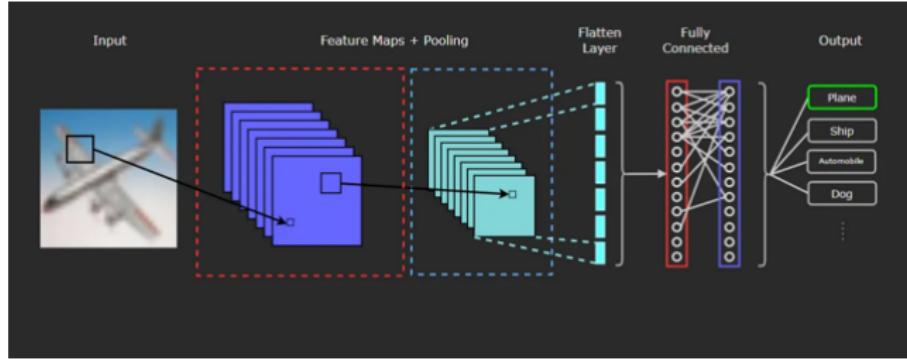
Finally, though the weights of the filters are the main parameters that are trained, there are also hyper-parameters that can be tuned for CNNs:

- number of filters in a layer
- dimensions of filters
- stride (number of pixels a filter moves each step)
- padding (how the filter handles boundaries of images)

We won't get into the details of these hyperparameters, since this isn't intended to be a comprehensive CNN walk-through, but these are important factors to be aware of.

# CCN Architecture

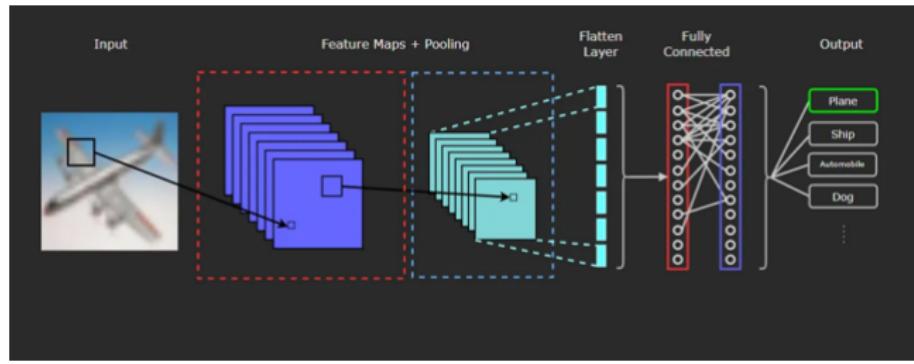
- The convolutional layer is only the beginning...
- In comparison to feed-forward networks, like the one we developed in the previous part of these lessons, CNN have different architecture, and **are composed of different types of layers**.
- In the figure below, we can see the general architecture of a typical CNN, including the different types of layers it can contain.



# CCN Architecture

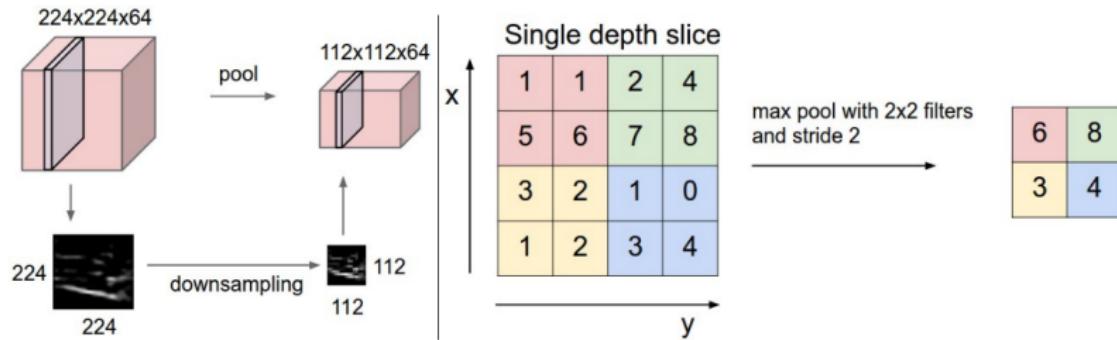
The three types of layers usually present in a Convolutional Network are:

- Convolutional Layers (red dashed outline)
- Pooling Layers (blue dashed outline)
- Fully Connected Layers (Red and Purple solid outlines)



# Pooling

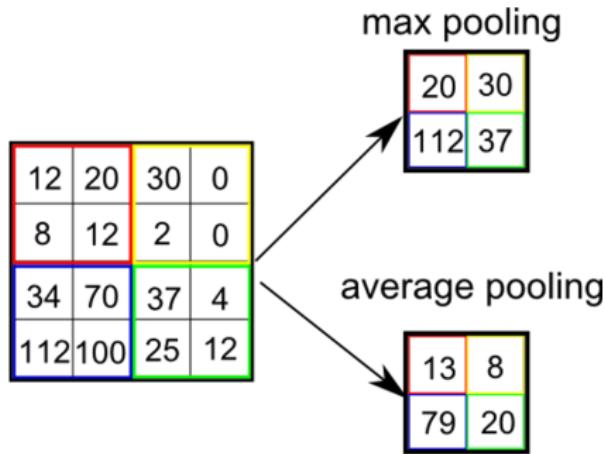
- Pooling layers are similar to convolutional layers, in that a filter convolves over the input data (usually a feature map that was output from a convolutional layer).
- However, rather than feature detection, the function of pooling layers is dimensionality reduction or downsampling.



Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. Left: In this example, the input volume of size  $[224 \times 224 \times 64]$  is pooled with filter size 2, stride 2 into output volume of size  $[112 \times 112 \times 64]$ . Notice that the volume depth is preserved. Right: The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little  $2 \times 2$  square).

# Pooling

- The two most common types of pooling used are Max Pooling and Average Pooling.
- With Max Pooling, the filter slides across the input, and at each step will select the pixel with the largest value as the output.
- In Average Pooling, the filter will output the average value of the pixels that the filter is passing over.



# CNN Output

- The last layer of a CNN is the classification layer which determines the predicted value based on the activation map.
- If you pass a handwriting sample to a CNN, the classification layer will tell you what letter is in the image.
- This is what autonomous vehicles use to determine whether an object is another car, a person, or some other obstacle.

# Training

- Training a CNN is similar to training many other machine learning algorithms.
- You'll start with some training data that is separate from your test data and you'll tune your weights based on the accuracy of the predicted values.
- Just be careful that you don't overfit your model.

## Subsection 2

### Sequential Data

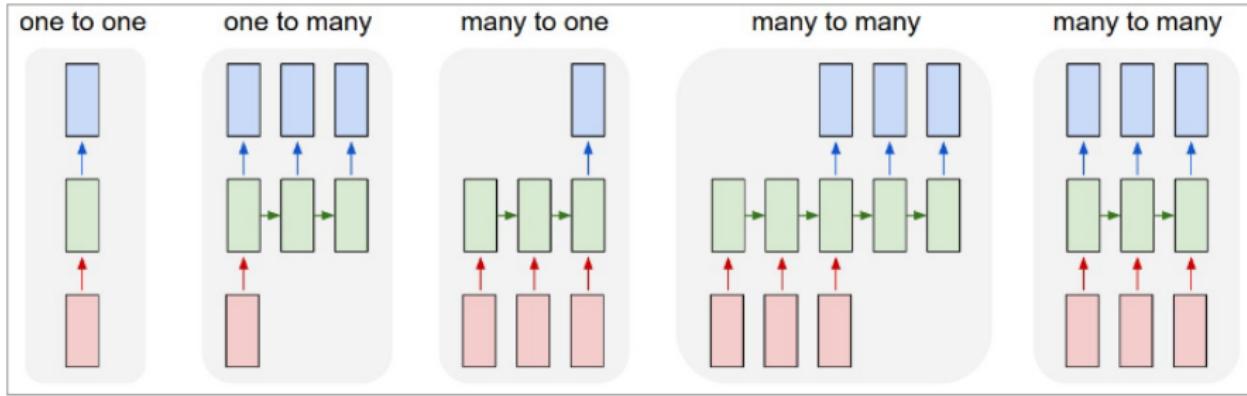
# What are Sequential Data

- Sequential data refers to data that has a natural ordering to it, meaning that **each data point is dependent on the data that came before it**. In other words, **the sequence of the data matters**.
- Examples of sequential data include time series data such as stock prices, weather data, or audio signals, where each point in time is dependent on the previous point. Other examples include text data such as sentences or paragraphs, where the meaning of each word is dependent on the context of the words that came before it.
- Sequential data can be modeled and analyzed using various techniques, including Hidden Markov Models (HMMs), and autoregressive models such as ARIMA.

# What are Sequential Data

- In the context of machine learning, sequential data presents unique challenges because the order of the data is important and traditional machine learning models such as decision trees and linear regression do not take this into account.
- Models such as RNNs (Recurrent Neural Network) are specifically designed to handle sequential data by processing the data one element at a time and retaining information about the previous elements to make predictions about future elements in the sequence.

# What are Sequential Data



Each rectangle is a vector and arrows represent functions (e.g. matrix multiply). Input vectors are in red, output vectors are in blue and green vectors hold the RNN's state (more on this soon). From left to right: (1) Vanilla mode of processing without RNN, from fixed-sized input to fixed-sized output (e.g. image classification). (2) Sequence output (e.g. image captioning takes an image and outputs a sentence of words). (3) Sequence input (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment). (4) Sequence input and sequence output (e.g. Machine Translation: an RNN reads a sentence in English and then outputs a sentence in French). (5) Synced sequence input and output (e.g. video classification where we wish to label each frame of the video). Notice that in every case are no pre-specified constraints on the lengths of sequences because the recurrent transformation (green) is fixed and can be applied as many times as we like.

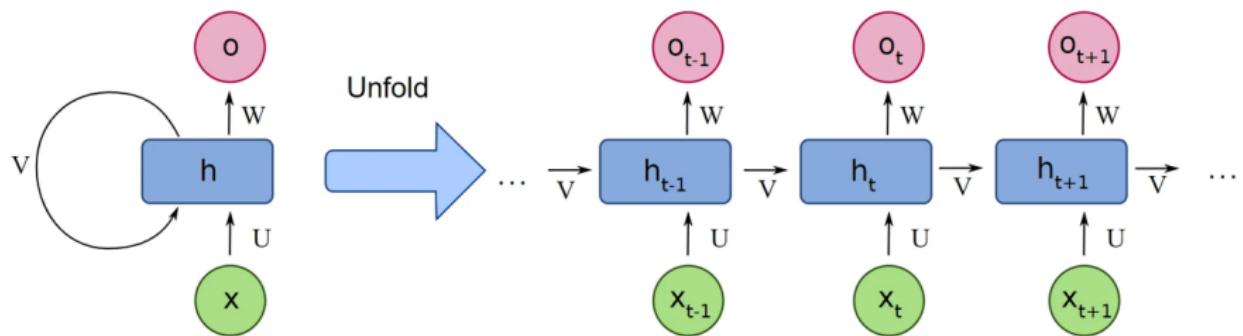
## Subsection 3

# Recurrent Neural Networks

# What are Recurrent Neural Networks

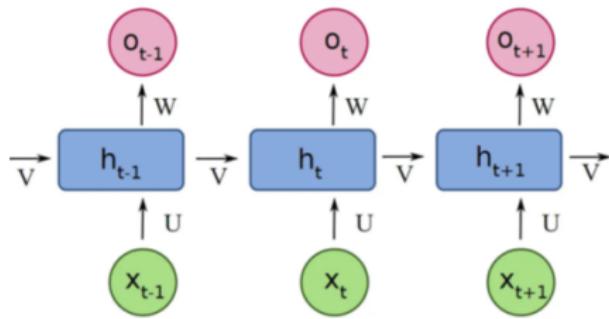
- A major characteristic of all neural networks you have seen so far, such as densely connected networks and convnets, is that they have no memory.
- Each input shown to them is processed independently, with no state kept in between inputs.

# What are Recurrent Neural Network



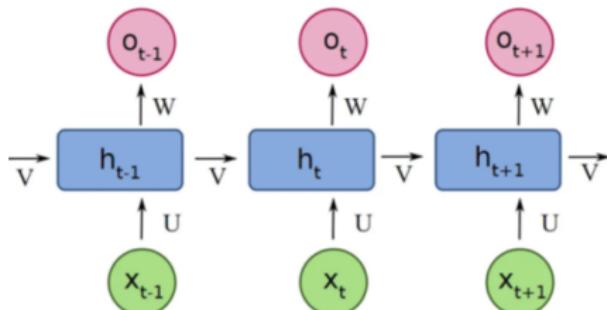
# What are Recurrent Neural Network

- In the picture, there are some distinct components, from which the most important are:
- $x$ : The input. It can be a word in a sentence or some other type of sequential data
- $O$ : The output. For instance, what the network thinks the next word on a sentence should be given the previous words



# What are Recurrent Neural Network

- In the picture, there are some distinct components, from which the most important are:
- **h**: The main block of the RNN. It contains the weights and the activation functions of the network
- **V**: Represents the communication from one time-step to the other.



# What are Recurrent Neural Networks

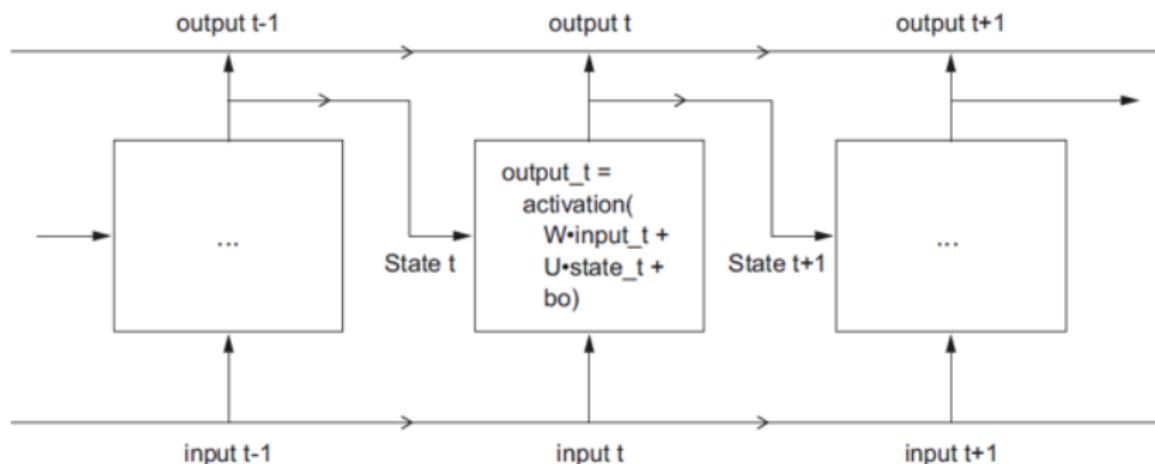


Figure 6.10 A simple RNN, unrolled over time

# What are Recurrent Neural Networks

- A recurrent neural network (RNN) processes sequences by iterating through the sequence elements and maintaining a state containing information relative to what it has seen so far.
- In effect, an RNN is a type of neural network that has an internal loop: **the network internally loops over sequence elements.**

## Listing 6.19 Pseudocode RNN

```
state_t = 0           ← The state at t
for input_t in input_sequence:   ← Iterates over sequence elements
    output_t = f(input_t, state_t)
    state_t = output_t   ← The previous output becomes the state for the next iteration.
```

# What are Recurrent Neural Networks

- The transformation of the input and state into an output will be parameterized by two matrices,  $W$  and  $U$ , and a bias vector.
- It is similar to the transformation operated by a densely connected layer in a feedforward network.

## Listing 6.20 More detailed pseudocode for the RNN

```
state_t = 0
for input_t in input_sequence:
    output_t = activation(dot(W, input_t) + dot(U, state_t) + b)
    state_t = output_t
```

# What are Recurrent Neural Networks

- In this very simple example, the final output is a  $2D$  tensor of shape (*timesteps, output\_features*), where each timestep is the output of the loop at time  $t$ .
- Each timestep  $t$  in the output tensor contains information about timesteps 0 to  $t$  in the input sequence about the entire past.
- For this reason, in many cases, you don't need this full sequence of outputs; you just need the last output (*output\_t* at the end of the loop), because it already contains information about the entire sequence.

# Recurrent layer in Keras

The SimpleRNN layer:

---

```
from keras.layers import SimpleRNN
```

---

SimpleRNN processes batches of sequences, like all other Keras layers.  
This means it takes inputs of shape (*batch\_size, timesteps, input\_features*),  
rather than (*timesteps, Input\_features*).

## Subsection 4

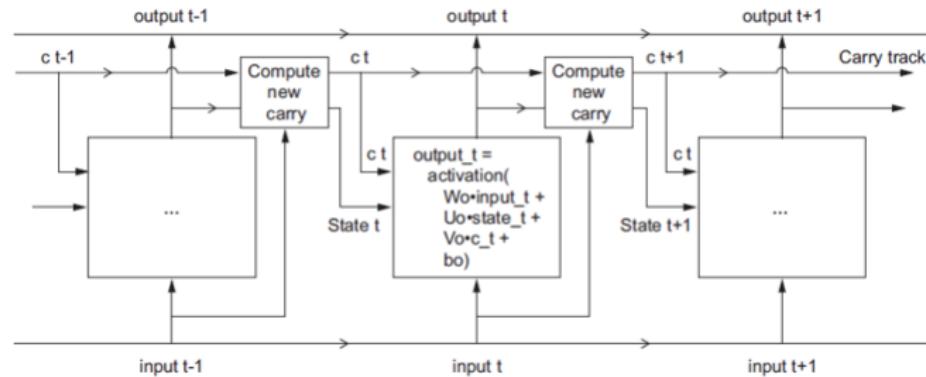
### Long short-term memory NN

# Problems with RNN

- SimpleRNN is not the only recurrent layer available in Keras.
- There are two others: **LSTM** and **GRU**.
- In practice, you will always use one of these, because SimpleRNN is generally too simplistic to be of real use.
- SimpleRNN has a major issue: although it should theoretically be able to retain at time  $t$  information about inputs seen many timesteps before, **in practice, such long-term dependencies are impossible to learn.**
- This is due to the **vanishing gradient problem**, an effect that is similar to what is observed with non-recurrent networks (feedforward networks) that are many layers deep: as you keep adding layers to a network, the network eventually becomes untrainable.
- The theoretical reasons for this effect were studied by Hochreiter, Schmidhuber, and Bengio in the early 1990s.
- The LSTM and GRU layers are designed to solve this problem.

# LSTM layer in Keras

- Let's consider the LSTM layer.
- This layer is a variant of the SimpleRNN layer you already know about; it adds a way to carry information across many timesteps.



# LSTM layer in Keras

Now the subtlety: the way the next value of the carry dataflow is computed. It involves three distinct transformations. All three have the form of a SimpleRNN cell:

---

```
y = activation(dot(state_t, U) + dot(input_t, W) + b)
```

---

# Pseudocode details of the LSTM architecture

---

```
output_t = activation(dot(state_t, Uo) + \
                      dot(input_t, Wo) + \
                      dot(C_t, Vo) + bo)
i_t = activation(dot(state_t, Ui) + dot(input_t, Wi) + bi)
f_t = activation(dot(state_t, Uf) + dot(input_t, Wf) + bf)
k_t = activation(dot(state_t, Uk) + dot(input_t, Wk) + bk)
#
# You obtain the new carry state (the next c_t) by combining
# i_t, f_t, and k_t
#
c_t+1 = i_t * k_t + c_t * f_t
```

---

# Credits and References

# Credits and References

- Renu Khandelwal, Convolutional Neural Network(CNN) Simplified
- <https://medium.datadriveninvestor.com/convolutional-neural-network-cnn-simplified-ecafd4ee52c5>

# Credits and References

- Nafiu, Stock market prediction using LSTM; will the price go up tomorrow. Practical guide  
(<https://medium.com/@nafiu.dev/stock-market-prediction-using-lstm-will-the-price-go-up-tomorrow-practical-guide-d1df2d54a517>)

# Exercises

<https://sh-tsang.medium.com/tutorial-a-good-toy-dataset-for-lstm-model-89e99063610c>